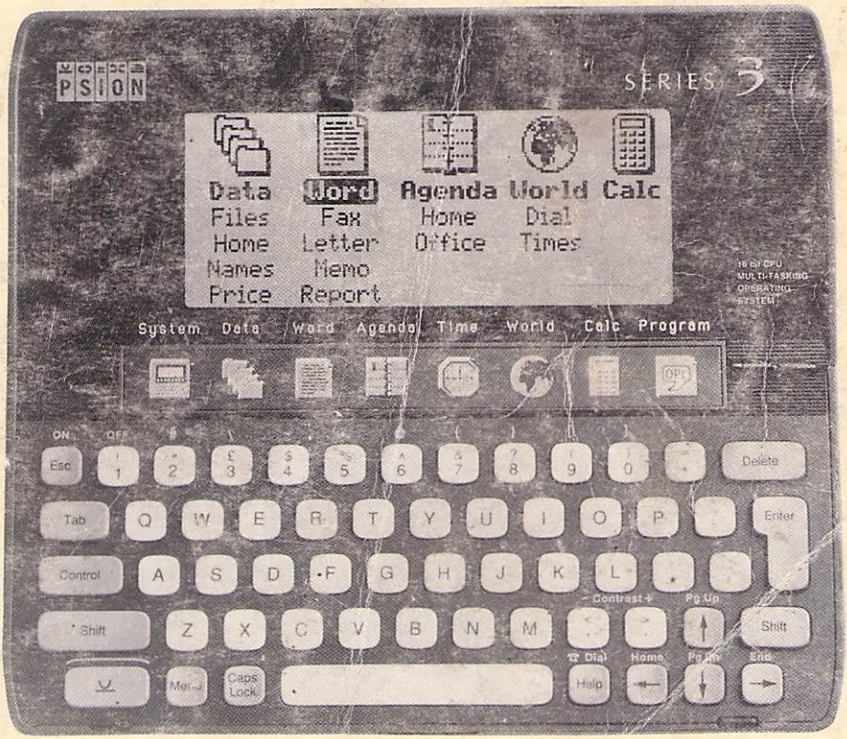
PSION SERIES 3

Programming Manual



PSION SERIES 3

Programming Manual

PSION

© Copyright Psion PLC 1991

All rights reserved. This manual and the programs referred to herein are copyrighted works of Psion PLC, London, England. Reproduction in whole or in part, including utilisation in machines capable of reproduction or retrieval, without the express written permission of Psion PLC is prohibited. Reverse engineering is also prohibited.

The information in this document is subject to change without notice.

Psion and the Psion logo are registered trademarks, and Psion Series 3, Psion HC, Psion MC, SSD and Solid State Disk are trademarks of Psion PLC.

IBM PC and IBM AT are registered trademarks of International Business Machines Corp. Apple and Macintosh are registered trademarks of Apple Computer Inc. Psion PLC acknowledges that some other names referred to are registered trademarks.

Version 2.0
Nov 91
Part no. 6103-0002-02
English

Contents

Introduction to OPL

1 *Creating and running programs* 3
Creating a new module 6
Module names 6
Type in and edit the procedure 7
Translating a module 9
Running a translated module 10
New OPL modules 11

2 *Variables and constants* 13
Declaring variables 15
Choosing the variable 15
Examples 18
Giving values to variables 19
Displaying variables 22
Values from the keyboard 23

3 *Loops and branches* 27
Repeating a set of instructions (loops) 29
Doing one set of instructions or another 31
The test conditions - 'True' and 'False' 33
Jumping to a different line of the program 34

4 *Calling procedures* 39
Using more than one procedure 40
Parameters 43
GLOBAL variables 46

5 *Data file handling* 49
Files, records and fields 51
Creating a data file 52
Opening a file 53
Saving records 55
Changing the current file 57
Moving from record to record 58
Finding a record 59
Data files and the Database 60

Reference

12 *Overview of functions and commands* 153
Summary, listed by task 156

13 *Alphabetic listing* 167

Appendices

A *Summary for users with OPL experience* . . . 227

B *Operators and logical expressions* 235

C *Printing from OPL* 243

D *Character codes* 249

Section 1

Introduction to OPL

The following chapters cover the basic concepts of OPL programming.

1

Creating and running programs

There are 3 stages to producing a program in OPL:

- ***Type in the program*, using the Program editor (the Program icon on the System screen).**

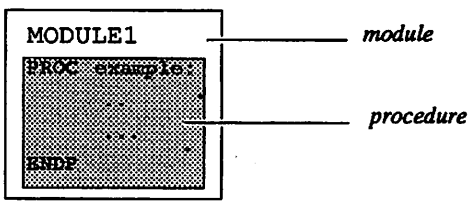
- ***Translate* the program. This makes a new version of your program in a format which the Series 3 can "run".**

- ***Run* the program. If it does not work as you had intended, re-edit it, then translate and run it again.**

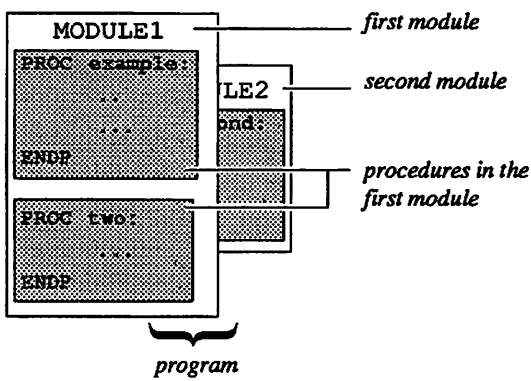
IMPORTANT:
This chapter guides you through these stages with a simple example. If you wish to follow the example, note that each instruction for you to do something is marked by a ➢ symbol.

Definitions

- A *procedure* is a single programming unit. Programs can be built up out of a number of procedures. A procedure starts with the keyword PROC and ends with the keyword ENDP. The rest of the procedure consists of other OPL keywords, used according to certain rules.
- A *module* is just the file itself - you say "OPL *module*" as you might say "Word Processor *document*". A module can contain one or more procedures.
- *Program* is the general word for the "finished product" - a set of procedures which can be run. A program may consist of one module containing one procedure:



or it may consist of a number of modules containing various procedures:



These terms do overlap, especially when dealing with single modules containing just one procedure.

The following chapters start from the basics of programming, and build up to more complex programs using a number of procedures.

Creating a new module

The next few pages work with this example procedure:

```
PROC test:
   PRINT "This is my OPL program"
   PAUSE 80
   CLS
   PRINT "Press a key to finish"
   GET
ENDP
```

This procedure does nothing of any real use - it is just an example of how some common OPL keywords work. The procedure first displays `This is my OPL program` on the screen. After a few seconds this is replaced by `Press a key to finish` . Then, when you press a key, the program finishes.

You can mix upper and lower case letters freely throughout OPL.

Module names

You should normally choose a name which describes what a module does. Then, when you've written several modules, you can still recognise which is which.

The name:

- Can be up to 8 characters long
- Can include numbers, but **must** start with a letter.

Examples:

```
test
shares93
taxcalc
```

Create a new module **now**:

➢ Move to the Program icon on the System screen, and select 'New file' from the 'File' menu. Type `test` as the name to use for this OPL module, then press Enter.

Type in and edit the procedure

You can now type in the procedure. The Program editor is very much like the Word Processor, but uses neither styles nor emphases. Text you type does not word-wrap; each line is a separate paragraph.

On the first line, you'll see `PROC :`, which marks the beginning of a procedure; `ENDP`, on line two, marks its end. You should type in the procedure between these two lines.

First, type a name for the procedure, after `PROC`. The flashing cursor is automatically in the correct place (before a colon) for you to do this. You can choose any name you want. For simple procedures which are the only procedure in a module, you often use the same name as the module.

➢ Type `test` . The top line should now read `PROC test:` .

➢ Press ↓. The cursor is already indented, as if the Tab key had been pressed.

You can now type the lines of the procedure:

➢ Type `PRINT "This is my OPL program"`. (Note the space after `PRINT`.) Press Enter at the end of the line.

Each new line is automatically indented, so you don't need to press the Tab key each time. (You don't have to indent lines, but it makes a procedure easier to understand.)

➢ Type the other lines of the procedure.

You can, if you want, combine lines of OPL, by separating each one with a space and colon - for example, the two lines:

```
PAUSE 80
CLS
```

could be combined as this one line:

```
PAUSE 80 :CLS
```

You can, of course, use the other Series 3 features at any time while you are editing. Use the Program button to return to editing your program.

What the different keywords do

PRINT - a *command*, in other words an instruction to OPL to do something. PRINT takes some text, in quote marks, and displays it on the screen. The text to be displayed is `This is my OPL program`.

PAUSE - another command, telling OPL to wait. `PAUSE 80` waits for 4 seconds. (`PAUSE 20` would wait for 1 second, `PAUSE 200` for 10 seconds, and so on.)

CLS - another command. It clears the screen.

GET - this is a *function*. A function is just like a command but it also gives a value which you can use. GET waits for you to press a key on the keyboard, and gives a value which identifies the key pressed. In this example the value is ignored, as GET is being used to provide a pause while you read the screen. This is a common use of the GET function.

All these lines are *statements* - separate instructions which OPL reads and acts upon. When the last statement in the procedure has been *executed* (acted upon) the procedure ends:

ENDP - marks the end of the procedure.

Menu options while editing

While you're typing in the procedure, all the editing facilities you use in the Word Processor - such as the 'Copy text' and 'Insert text' options - are available. Refer to the chapter on the Word Processor in the User Guide for more information.

The menus available are the same as in the Word Processor, except that the 'Word' menu has been replaced by a 'Prog' menu. This has options to 'Translate' the current program, and to 'Run' a translated program. It also has a 'Show error' option, to re-display an error which prevented successful translation, and an 'Indentation' option, for setting the tab width and to turn auto-indentation on and off.

The 'Outline' option lists only the names of each procedure, for quick navigation around an OPL module. (The Program button does this too.)

The 'New file' option will only ever use a template called 'default'. When you select 'Save as template', you are given the option of replacing this template. (If you say 'No' to this, it is then possible to save the file as a template under a different filename.) **Do not try to mix templates between the Word Processor and the Program editor.**

'Set preferences' has settings for bold/normal and mono/proportional text.

There is no 'Password' option.

Translating a module

This makes a separate version of your program in a format which the Series 3 can *run*.

You'd usually try to translate a module as soon as you finish typing it in, to check for any typing mistakes you've made, and then to see if the program runs as you intended.

After translating a module, you can choose whether or not to run it.

➢ Select the 'Translate' option from the 'Prog' menu.

What happens when you translate a module?

First: the procedures in the module are checked for errors.

If the Series 3 cannot understand a procedure, because of a typing error, a message is shown, such as 'Syntax error'. The cursor is positioned at the point where the mistake is, so that you can correct it. For example, you might have typed `PRONT "This is..."`, or `PAUSE80` without the space.

When you think you've corrected the mistake, select 'Translate' from the 'Prog' menu again. If there are still mistakes, you are again taken back to where they occur.

There may still be errors in your program, however, since there are some errors which cannot be detected until you try to run the program.

If you've already used up almost all of the memory, the Series 3 may be unable to translate the program, and report a 'Memory full' message. You'll need to free some memory, as described in the 'Troubleshooting' chapter of the User Guide, before trying again.

When there are no more errors, a separate version of your program is produced, in a format which the Series 3 can run.

Running a translated module

When you've successfully translated a module, you can run it. You'd usually run it straight away in order to test it.

Running a module does require some free memory, so again a 'Memory full' message is possible.

Running after translating

➢ When your module translates successfully, a dialog is displayed, asking whether to run the translated module. Accept this by pressing 'Y'; the screen is cleared, and the module runs.

When the module has finished running, you return to the Program editor, with the cursor where it was before.

If an error occurs while the module is running, you will return to editing the module, with the cursor at the point where the error occurred.

Running again

You can run a module again, if you wish, by selecting 'Run' from the 'Prog' menu. This runs the **translated** version of your program; if you've made changes but you haven't translated it again, they'll have no effect.

'Run' displays a dialog, letting you type the name of **any** translated module which you want to run.

Running from the System screen

The names of any successfully translated programs automatically appear under a new icon in the System screen. This is at the right-hand end of the list of icons (past the Program icon), and is usually off the right-hand edge of the screen. The icon is just the word `OPL` in a speech bubble, and is called the RunOpl icon.

➢ Press the System button to go to the System screen. Move the cursor right to the RunOpl icon, onto the program name `test`, and press Enter.

You can also press Control-Program (hold down Control and press the Program button) as a short-cut to move to the RunOpl icon, whatever you're doing. (If there is a running program, this instead moves **directly** to it.)

When an OPL program is finished and working, you will usually run it from the System screen. But while you're still editing and testing, it's quicker to run it from inside the Program editor. This also positions the cursor for you, if errors occur.

Stopping a program while it's running

To stop a running program, press PSION-Esc.

You can pause a running program by pressing Control-S. It will be paused as soon as it next tries to display something on the screen. Press any other key to let the program resume running.

New OPL modules

You can create new OPL modules in the same way as new Word Processor documents. Use the 'New file' option in the Program editor, or move to the Program icon in the System screen and use its 'New File' option.

Your module names are listed below the Program icon. The word 'Program' is only shown below the icon if there are no modules at all.

The names under the RunOpl icon are those modules which have been translated successfully.

To re-edit an existing OPL program, use the 'Open file' option in the Program editor, or move to the Program icon in the System screen and select it from the list.

Copying modules

Use the 'Copy file' option in the System screen to copy modules (or translated modules). See the User Guide for full details. You can also use the 'Save as' option in the Program editor itself.

Deleting modules

You can delete an OPL module as you would any other file - go to the System screen, move the highlight onto the file and use the 'Delete file' option.

★ *The status window*

A temporary status window is always available while an OPL program is running. Press PSION-Menu to see it.

★ *Looking at a running program*

If you translate and run a module from the Program editor, the Program button will still return to the editor, even if the translated program has not finished running. A 'Busy' message is shown - you can move the cursor around the program as normal, but you can't edit it.

To return to the running version, select it from beneath the RunOpl icon in the System screen. It will be in bold, at the top of the list, to show that it is currently running. Alternatively, press Control-Program.

★ *The RunOpl icon*

If you delete all of your translated modules, the RunOpl icon will remain on the System screen, with the word 'RunOpl' beneath it.

★ *Running more than module*

If a module is running, and you select a second one from the System screen, the first one is **not** replaced - both modules run together, and will be in bold on the file list. Control-Shift-Program swaps between them.

Although you may never need to, you **can** run more than one copy of the same translated OPL module **at the same time**. There are two ways:

- Use 'Copy file' in the System screen to make a new copy of the module, with a different filename. Then run both files.
- Run the file as normal. Then move the highlight to the file list under the RunOpl icon, press Tab to show the file selector, and pick the name of the translated module again.

★ *SUMMARY*

Move to the Program icon in the System screen and select the 'New file' option.

Type in your procedure.

Select 'Translate' from the 'Prog' menu.

When a module translates correctly you are given the option to run it. You can run it again at any time, either with 'Run' in the 'Prog' menu, or directly from the RunOpl icon in the System screen.

2

Variables and constants

Programs can process data in a variety of ways. They may, for example, perform calculations with numbers, or save and recall *strings* of text (such as names and phone numbers in a data file).

In all cases, your program must be able to handle *values* - different types of numbers, strings, and so on.

In OPL, there are two ways of handling values: *variables* and *constants*. Constants are fixed values, such as 1, 2, 3. Variables are used to store values which may change - for example, a variable called X may start with the value 3 but later take the value 7.

Note
The example programs in this and the following few chapters do not include full error handling. This has been done in order to keep the programs short and easy to understand. But it means that, when you run one of these programs and, for example, fail to type in the kind of value which the program asks for, the program may fail - harmlessly stopping before it completed.

You may want to develop your own programs from the examples in the following chapters. In this case, it is recommended that you add some error handling code to the programs. A later chapter gives a full explanation of error handling.

Declaring variables

Most procedures begin by *declaring* (creating) variables:

```
LOCAL x,y,z
```

LOCAL is the word telling the Series 3 to create variables, with the names which follow - here x, y and z - separated by commas.

The statement `LOCAL x,y,z` defines three variables called x, y and z. The Series 3 will recognise these names whenever you use them in this procedure. (If you used them in another procedure, they wouldn't be recognised; the variables are 'local' to the procedure in which they are declared.)

These variables are initially given the value 0.

Any variables you wish to use must be declared at the **start** of a procedure.

Choosing the variable

Before declaring variables, decide what information they are going to contain. There are different types of variables for different sorts of values. If you try to give the wrong type of value to a variable, an error message will be displayed.

You specify the type of each variable when you declare it, by adding a symbol at the end of its name.

Numbers

- For small whole numbers - for example `6` - use an ***integer variable.*** Integer variables have a `%` symbol on the end, for example `number%`.

  Integer variables can handle numbers only in the range -32768 to +32767. If you try to give an integer variable a whole number bigger than this, an error message will be displayed.

- For larger whole numbers - for example `10000000` - use a ***long integer variable.*** Long integer variables have an `&` symbol on the end, for example `number&`.

Long integer variables can handle whole numbers in the range -2147483648 to +2147483647. If a variable may have to handle numbers outside normal integer range, make it a long integer variable.

- For non-whole numbers - for example `2.5` - use a ***floating-point variable***. Floating-point variables have no symbol on the end - `price`, for example.

  If you know that at some stage in your program your variable will have to handle non-whole numbers, like `1.2`, use a floating-point, not an integer variable. Otherwise you may get unpredictable results. (There's more about this later in this chapter.)

- For very large numbers - outside long integer range - you should also use floating-point variables. These can handle numbers as big as `±9.99999999999e99` and as small as `±1e-99`. (Intermediate results in calculations may even be up to $\pm 10^{307}$.)

Text

For text - `Are you sure?`, `54th`, etc. - use a ***string variable***. (Pieces of text are called *strings* in OPL.) String variables have a `$` symbol on the end - for example, `name$`.

To declare a string variable, you **must** follow the `$` symbol with the maximum length of string you want the variable to handle, in brackets. So if you want to store names up to 15 characters long in the variable `NAME$`, declare it like this: `LOCAL NAME$(15)`. Strings cannot be longer than 255 characters.

Array variables

You may want a group of variables, for example to store lists of values. Instead of having to declare separate variables `a`, `b`, `c`, `d` and `e`, you can declare *array* variables `a(1)` to `a(5)` in one go like this:

```
LOCAL a%(5)      (array of integer variables)
LOCAL a(5)       (array of floating-point variables)
LOCAL a$(5,8)    (array of string variables)
or
LOCAL a&(5)      (array of long integers)
```

The number in brackets is the number of elements in the array. So `LOCAL a%(5)` creates five integer variables: `a%(1)`, `a%(2)`, `a%(3)`, `a%(4)` and `a%(5)`.

With strings, the second number in the brackets specifies the maximum length of the strings. All the elements in the string array have the same

capacity - for example, `LOCAL ID$(5,10)` allocates memory space for five strings, each up to ten characters in length.

OPL does not support two-dimensional arrays.

Initial values

All numeric variables have zero as their initial value. String variables have a string with no characters in it. Every element in an array variable is also initialised in the appropriate way.

Choosing descriptive names

To make it easier to write your programs, and understand them when you read through them at a later date, give your main variables names which describe the values they hold. For example, in a procedure which calculates fuel efficiency, you might use variables named `speed` and `distance`.

All variable names:

- May be up to 8 characters long
- **Must** start with a letter, but after that may use any combination of numbers and letters
- May be entered in any combination of upper and lower case. `sPeeD` and `SpEEd` would be considered as the same name.

Additionally, you must not use any of the function or command names listed in the Reference section of this manual - if you use these you will see a 'Declaration error' message when you translate your module.

The `$` `&` and `%` symbols *are included* in the 8 characters allowed in variable names - so `V2345678%` is too long to be a valid variable name, but `V234567%` is acceptable.

Examples

- `LOCAL clients$(12),z&(3)` declares one string variable, `clients$`, of capacity twelve characters, and one long integer array variable containing three elements, `z&(1)`, `z&(2)` and `z&(3)`
- `LOCAL AGE%,B5$(10),i` declares one integer variable, `AGE%`, one string variable, `B5$`, of capacity ten characters, and one floating-point variable, `i`
- `LOCAL profit93` declares one floating-point variable, `profit93`
- `LOCAL x,MAN6$(4,7)` declares one floating-point variable, `x`, and one string array variable, `man6$`, containing four elements, `man6$(1)`, `man6$(2)`, `man6$(3)` and `man6$(4)`, each of capacity 7 characters

For preference

- Integer variables use less memory than long integer variables, and both use less than floating-point.
- Integer variables are processed faster than floating-point.

Giving values to variables

Assigning values

You can *assign* a value to a variable directly, like this:

```
x=5
y=10
```

So a procedure to add two numbers together could look like this:

```
PROC add:
   LOCAL x%,y%,z%
   x%=569
   y%=203
   z%=x%+y%
   PRINT z%
   GET
ENDP
```

`add:` is the procedure name.

The `LOCAL` statement defines three variables `x%`, `y%` and `z%`, all initially with the value 0. `PRINT` displays the value of `z%` on the screen. You can display the value of any variable like this.

`PROC` and `ENDP` define the beginning and end of the procedure - as you saw in Chapter 1.

Assigning values to string variables

String variables can be assigned text values like this:

```
a$="some text"
```

The text to use must be enclosed in double quote characters.

Assigning values to an array variable

If you declare a%(4), assign values to each of the elements in the array like this: `a%(1)=56, a%(2)=345` and so on. Similarly for the other variable types: `a(1)=.0346, a&(3)=355440, a$(10)="name"`.

Arithmetic operations

You can use these *operators*:

+	plus
-	minus or make negative
/	divide
*	multiply
**	raise to a power
%	percentage

For more information on the operators, see Appendix B.

Expressions

You can assign a value to a variable with an *expression* - that is, a combination of numbers, variables, and functions. For example:

`z=x+y/2` gives the `z` the value of `x` plus the value of ½.

`z=x*y+34.78` gives `z` the value of `x` times `y`, plus `34.78`.

`z=COS(y)` gives `z` the value of the cosine of `y`. COS is an OPL *function*. Unlike the GET function, COS requires a value or variable to work with. As you can see, you put this in brackets, after the function name.

These are all *operations* using the variables `x` and `y` - assigning the result to `z` and not actually affecting the value of `x` or `y`. The left hand side of expressions must always be a simple variable name.

The ways you can change the values of variables fall into these groups:

- Arithmetic operations, such as multiplication or addition - for example `z=sales+costs` or `z=y%*(4-x%)`
- Using one of the OPL functions, for example `z=SIN(PI/6)`

or

- Using keywords like INPUT or EDIT which wait for you to type in values from the keyboard.

Self reference
Arithmetic operations can include self-reference. For example:

`z%=z%+1` (make the value of `z%` one greater than its current value)

`x%=y+x%/4` (make the value of `x%` a quarter of its current value, plus the value of `y`)

Precedence of operators

Operators have the same precedence as in the Calculator. For example, `3+51.3/8` is treated as `3+(51.3/8)`, not `(3+51.3)/8`. The table of precedence is repeated in appendix B.

Constants

In an OPL program, numbers (and strings in quote marks) are sometimes called *constants*. In practice, you will use constants without thinking about them. For example:

```
x=0.32
x%=569
x&=32768
x$="string"
x(1)=4.87
```

OPL can also represent hexadecimal constants. This is explained under the HEX$ entry in the 'Alphabetic listing' chapter.

Exponential notation may be useful for very large or very small numbers. Use E (capital or lower case) to mean "times ten to the power of" - for example, 3.14E7 is $3.14*10^7$ (31400000), while 1E-9 is $1*10^{-9}$ (0.000000001).

Problems with integers

When calculating an expression, OPL uses the simplest arithmetic possible for the numbers involved. If all of the numbers are integers, integer arithmetic is used; if one is outside integer range but within long integer range, then long integer arithmetic is used; if any of the numbers are not whole numbers, or are outside long integer range, floating-point arithmetic is used.

This has the benefit of maximising speed, but you must beware of calculations going out of the range of the type of arithmetic used. For example, in `X=200*300` both `200` and `300` are integers, so integer arithmetic is used for speed (even though `X` is a floating-point variable). However, the result, 60000, cannot be calculated because it is outside integer range (32767 to -32768), so an 'Integer Overflow' error is produced.

You can get around this by using the INT function, which turns an integer into a long integer, without changing its value. If you rewrite the previous example as `X=INT(200)*300`, OPL has to use long integer arithmetic, and can therefore give the correct result (60000). (If you understand

hexadecimal numbers, you can instead write one of the numbers as a hexadecimal long integer - e.g. `200` would become `&C8`.)

Integer arithmetic uses whole numbers only. For example, if `y%` is 7 and `x%` is 4, `y%/x%` gives 1. However, you can use the INTF function to convert an integer or long integer into a floating-point number, forcing floating-point arithmetic to be used - for example, `INTF(y%)/x%` gives 1.75.

If one of the integers in an all-integer calculation is a constant, you can instead write it as a floating-point number. 7/4 gives 1, but 7/4.0 gives 1.75.

Operations on strings

If `a$` is `"down"` and `b$` is `"wind"`, then the statement `c$=a$+b$` means `c$` becomes `"downwind"`.

Alternatively, you could give `c$` the same value with the statement `c$="down"+"wind"`.

When adding strings together, the result must not be longer than the maximum length you declared - e.g. if you declared `LOCAL a$(5)` then `a$="first"+"second"` would cause an error to be displayed.

+ is the only operator that works on strings. To cut up strings, use string functions such as MID$, LEFT$ and RIGHT$, explained in a later chapter. You must use them to extract even a single character - you **cannot**, for example, refer to the fourth character in `a$(7)` by using `a$(4)`.

Displaying variables

PRINT is one of the most useful OPL commands. Use it to display any combination of text messages and the values of variables.

Where the cursor goes

In general, each PRINT statement ends by moving to a new line. For example:

```
A%=127 :PRINT "A% is"
PRINT a%
```

would display as

```
A% is
127
```

You can stop a PRINT statement from moving to a new line by ending it with a semi-colon. For example:

```
A%=127 :PRINT "A% is";
PRINT a%
```

would display as

```
A% is127
```

If you end a PRINT statement with a comma, it stays on the same line but displays an extra space. For example:

```
A%=127 :PRINT "A% is",
PRINT a%
```

would display as

```
A% is 127
```

▮ *Displaying a list of things*

You can use commas or semicolons to separate things to be displayed on one line, instead of using one PRINT statement for each. They have the same effect as before:

```
A%=127 :PRINT "A% is",a%
```

would display as

```
A% is 127
```

while

```
user$="Fred"
PRINT "Hello",user$;"!"
```

would display as

```
Hello Fred!
```

Values from the keyboard

If you want a program to be reusable, it often needs to be able to accept different sets of information each time you use it. You can do this with the INPUT command, which takes values typed in at the keyboard and assigns them to variables.

For example, this simple procedure converts from degrees centigrade to degrees fahrenheit. It waits for you to type in a number. You can edit it as you type - Delete, for example, deletes characters, and Esc clears everything you've typed. Press Enter when you've finished. The value is assigned to the variable `cent` and the result of the conversion is then displayed:

```
PROC temp:
   LOCAL cent
   AT 1,4
   PRINT "Temperature in centigrade?",
   INPUT cent  :REM value from keyboard
   PRINT "That's",32+cent*9/5,"fahrenheit."
   GET
ENDP
```

Here PRINT is used to show a message before the INPUT, saying what information needs to be typed in - here, 'Temperature in centigrade?'. Note the comma at the end of this statement, so that the INPUT cursor is on the same line as the message.

The value entered at the INPUT must be of the appropriate kind for the variable to which the value is to be assigned. INPUT will show a ? for you to enter another value if you entered the wrong type (for example, if you entered the string `ninety` for the floating-point variable `cent`). You can enter numbers of any type for assignment to a numeric variable - for example, 4 for the variable `x` - providing the number is within the numeric type's range.

Comments

The REM command lets you add comments to a program to help explain how it works. Begin the comment with the word REM (short for 'remark'). Everything after the REM command is ignored.

If you put a REM command on the end of a line, the colon you would normally put before it is optional. For example, you could use either of these:

```
CLS :REM Clears the screen
```

or

```
CLS REM Clears the screen
```

AT command

This positions the cursor or your message at the co-ordinates you specify. Use the command like this:

```
AT column%,row%
```

where `column%` and `row%` give the character position to use.

`AT 1,1` positions the cursor to the top left corner.

▮ *Single keypresses*

In addition to using INPUT to ask for values, your program can ask for single keypresses. Use one of these functions:

- GET waits for a keypress and returns the key pressed.
- KEY returns a key if any was pressed, but doesn't wait for one.

Every separate letter, number or symbol has a number which represents it, called a *character code*. The full list of character codes - the *character set* - is included as an appendix to the User Guide. GET and KEY return the character code of the key pressed - for example, if A were pressed, these functions would return the value 65. KEY returns 0 if no key was pressed.

KEY$ and GET$ work in the same way as KEY and GET, except that they return the key pressed as a string, not as a character code:

- GET$ waits for a keypress and returns the key pressed, as a string.
- KEY$ returns a key if any was pressed, but doesn't wait for one. KEY$ returns a null string ("") if no key was pressed.

Unlike INPUT, these functions do not display the key pressed on the screen, and do not wait for you to press Enter.

★ *Example using GET$*

```
PROC kchar:
   LOCAL k$(1)
   PRINT "Press a key, A-Z:"
   k$=GET$
   PRINT "You pressed",k$
   PAUSE 60
ENDP
```

As you'll see in the next chapter, single keypresses are useful for making decisions. A program might, for example, offer a set of choices which you choose from by typing the word's first letter, like this:
`Add (A) Erase (E) or Copy (C) ?`
Or it might ask for confirmation of a decision, by displaying a `YES or NO?` message and waiting until `Y` or `N` is pressed.

▮ *Modifier keys*

If you need to check for the Shift, Control, PSION and/or Caps Lock keys being used, see the description of the KMOD function, in the 'Alphabetic listing' chapter.

★ *The calculator memories*

The calculator memories M0 to M9 are available as floating-point variables in OPL. You might use them to allow OPL access to results from the calculator, particularly if you use OPL procedures from within the calculator.

It's best not to use them instead of declaring your own variables. Your OPL program might go wrong if another running OPL program uses them, or if you use them yourself in the calculator.

★ *SUMMARY*

Declare variables with one or more LOCAL statements in the line after *PROC*:

- ***Integer*** variables - for example `year%`
- ***Floating-point*** variables - for example `price`
- ***String*** variables - for example `name$(12)` where the maximum length is given in the brackets
- ***Long integer*** variables - for example `profit&`

Variables will be floating-point unless you add a symbol to the end of the variable name.

- ***Array*** variables - for example `prices%(4)` or `clients$(5,12)` where the first number inside the brackets specifies the number of elements, and the second number in the brackets, in the case of string arrays, specifies the maximum length.

Assign values to variables:

- Expressions - for example `x=5.5/y , profit=x-y`
- INPUT command - for example `INPUT a$`
- 'Add' strings - for example `a$="MR"+names$`

REM allows you to add comments to a program.

AT positions the cursor.

GET and KEY return the key pressed as a character code.
GET$ and KEY$ return the key pressed as a single-character string.
GET and GET$ wait until a key is pressed, KEY and KEY$ do not.

3

Loops and branches

Every program in the earlier chapters runs in a straight line from start to finish. Each consists of a number of instructions which are executed in the order they appear in the procedure.

To carry out an instruction more than once you could write it several times. That may be inefficient. A better method is for the procedure to automatically repeat a particular part as many times as you require. Or, you may want your program to do different things depending on the circumstances. There are a number of ways of doing these in OPL:

- **Repeating a set of instructions (called loops)**
- **Doing one set of instructions or another (called IF statements)**
- **Jumping from one line of your program to another**

Repeating a set of instructions (loops)

The DO...UNTIL and WHILE...ENDWH commands are *structures* - they don't actually do anything to your data, but control the order in which other commands are executed:

- DO...UNTIL repeats a set of instructions until a certain condition is true.
- WHILE...ENDWH repeats a set of instructions so long as a certain condition is true.

The *test condition* is at the end of the DO...UNTIL loop, but at the beginning of the WHILE...ENDWH loop.

DO...UNTIL

```
PROC test:
   LOCAL a%
   a%=10
   DO
      PRINT "A=";a%
      a%=a%-1
   UNTIL a%=0
   PRINT "Finished"
   GET
ENDP
```

The instruction DO says to OPL:

"Execute all the following instructions until an UNTIL is reached. If the condition following UNTIL is not met, repeat the same set of instructions until it is."

The first time through the loop, a%=10. 1 is subtracted from a%, so that a% is 9 when the UNTIL statement is reached. Since a% isn't zero yet, the program returns to DO and the loop is repeated.

a% goes down to 8, and again it fails the UNTIL condition. The loop therefore repeats 10 times until a% does equal zero.

When a% equals zero, the program continues with the instructions after UNTIL.

DO...UNTIL loops are always executed at least once.

▮ WHILE...ENDWH

```
PROC test2:
   LOCAL a%
   a%=10
   WHILE a%>0
      PRINT "A=";a%
      a%=a%-1
   ENDWH
   PRINT "Finished"
   GET
ENDP
```

The instructions between the WHILE and ENDWH statements are executed only if the condition given by the WHILE statement is true - in this case if a% is greater than 0.

Initially, a%=10 and so A=10 is displayed on the screen. a% is then reduced to 9. a% is still greater than zero, so A=9 is displayed. This continues until A=1 is displayed. a% is then reduced to zero, and so Finished is displayed.

Unlike DO...UNTIL, it's possible for the instructions between WHILE and ENDWH not to be executed at all.

★ Example using WHILE...ENDWH

```
PROC newkey:
   LOCAL k$(1)
   WHILE KEY :ENDWH
   PRINT "Press a new key."
   k$=GET$
ENDP
```

This procedure ignores any keys which may already have been typed, then waits for a new keypress.

KEY returns the value of a key that was pressed, or 0 if no key has been pressed. WHILE KEY :ENDWH reads any keys previously pressed, one by one, until they have all been read and KEY returns zero.

Doing one set of instructions or another

In a program, you might have several possible cases (x% may be 1, or it may be 2, or 3...) and want to do something different for each one (if it's 1, do this - but if it's 2, do that...). You can do this with the IF...ENDIF structure:

```
IF condition1
   do these statements
ELSEIF condition2
   do these statements
ELSEIF condition3
   do these statements
.
.
ELSE
   do these statements
ENDIF
```

These lines would do **either**

- the statements following the IF line (if condition1 is met)

or

- the statements following one of the ELSEIF lines (if one of condition2, condition3... is met)

or

- the statements following the ELSE line (if none of condition1, condition2, condition3... have been met).

You can cater for as many cases as you like with ELSEIF statements. You don't have to have any ELSEIFs. There may be either one ELSE statement or none; you do not specify conditions for the ELSE statement.

Every IF in your program must be matched by an ENDIF - otherwise you'll see an error message when you try to translate the module. The structure must start with an IF and end with an ENDIF.

★ "Nesting" loops - the 'Too complex' message

You can have up to 8 DO...UNTIL, WHILE...ENDWH and/or IF...ENDIF structures nested within each other. If you nest them any deeper, a 'Too complex' error message will be displayed.

★ Example using IF

```
PROC zcode:
   LOCAL g%
   PRINT "Are you going to press Z?"
   g%=GET
   IF g%=%Z OR g%=%z
      PRINT "Yes!"
   ELSE
      PRINT "No."
   ENDIF
   PAUSE 60
ENDP
```

% operator

The program checks character codes with the % operator. `%a` returns the code of `a`, `%Z` the code of `Z` and so on. Using `%A` is entirely equivalent to using 65, the actual code for `A`, but it saves you having to look it up, and it makes your program easier to follow.

Be careful not to confuse character codes like these with integer variables.

OR operator

OR operator OR lets you check for either of two conditions. OR is an example of a *logical operator*. There is more about logical operators later in this chapter.

★ Example using DO...UNTIL and IF

```
PROC testny:
   DO
      g$=UPPER$(GET$)
   UNTIL g$="N" OR g$="Y"
   IF g$="N"
      ... REM 'N' pressed
   ELSE
      ... REM 'Y' pressed
   ENDIF
ENDP
```

This procedure checks for a 'Y' or 'N' keypress. You'd put your own code in the IF statement, where . . . has been used.

Arguments

Some functions, as with commands like PRINT and PAUSE, require you to give a value or values. These values are called *arguments*. The UPPER$ function needs you to specify a string argument, and returns the same

string but with all letters in upper case. For example, `UPPER("12.+aBcDeF")` returns `"12.+ABCDEF"`.

Functions as arguments to other functions
Since GET$ returns a string, you can use this as the argument for UPPER$. `UPPER$(GET$)` waits for you to press a key, because of the `GET$`; the `UPPER$` takes the string returned and, if it's a letter, returns it in upper case. This means that you can check for `"Y"` without having to check for `"y"` as well.

The test conditions - 'True' and 'False'

The test condition used with `DO...UNTIL`, `WHILE...ENDWH` and `IF...ENDIF` can be any expression, and may include any valid combination of operators and functions. Examples:

Condition	*Meaning*
`x=21`	does the value of `x` equal 21? (Note - as this is a test condition, it does **not** assign `x` the value 21)
`a%<>b%`	is the value of `a%` not equal to the value of `b%`?
`x%=(y%+z%)`	is the value of `x%` equal to the value of `y%+z%`? (does **not** asssign the value `y%+z%` to `x%`).

The expressions actually return a *logical value* - that is, a value meaning either 'True' or 'False'. Any non-zero value is considered 'True' (to return a 'True' value, OPL uses -1), while zero means false. So if `a%` is 6 and `b%` is 7, the expression `a%>b%` will return a zero value, since `a%` is **not** greater than `b%`.

These are the conditional operators:

<	less than	<=	less than or equal to
>	greater than	>=	greater than or equal to
=	equal to	<>	not equal to

Logical operators

The operators AND, OR and NOT allow you to combine or change test conditions. This table shows their effects. (`c1` and `c2` represent conditions.)

Example	*Result*	*Integer returned*
`c1 AND c2`	True if both `c1` and `c2` are true	-1
	False if either `c1` or `c2` are false	0
`c1 OR c2`	True if either `c1` or `c2` is true	-1
	False if both `c1` and `c2` are false	0
`NOT c1`	True if `c1` is false	-1
	False if `c1` is true	0

However, AND, OR and NOT become *bitwise operators* - something very different from logical operators - **when used exclusively with integer or long integer values.** If you use `IF A% AND B%`, AND acts as a bitwise operator, and you may not get the expected result. You would have to rewrite this as `IF A%<>0 AND B%<>0`.

Operators, including bitwise operators, are discussed further in an Appendix to this manual.

Jumping to a different line of the program

Jumping out of a loop: BREAK

The BREAK command jumps out of a DO...UNTIL or WHILE...ENDWH structure. The line after the UNTIL or ENDWH statement is executed, and the lines following are then executed as normal. For example:

```
DO
   .....
   IF a=c
      BREAK
   ENDIF
   .....
UNTIL a=b
x%=3
..
```

Jumping to the test condition: CONTINUE

The CONTINUE command jumps from the middle of a loop to its test condition. The test condition may be the UNTIL line of a DO...UNTIL loop or the WHILE line of a WHILE...ENDWH loop. For example:

```
WHILE a<b
   ...
   IF a=c
      CONTINUE
   ENDIF
   ...
   .....
ENDWH
..
```

▮ *Jumping to a 'label': GOTO*

The GOTO command jumps to a specified *label*. Labels can be put anywhere in your procedure. In this example, when the program reaches the GOTO statement, it jumps to the label `exit::`, and continues with the line after it.

```
GOTO exit
PRINT "MISS THIS LINE"
PRINT "AND THIS ONE"
exit::
```

- the two PRINT statements are missed out.

Labels themselves must end in a double colon. This is optional in the GOTO statement - both GOTO `exit::` and GOTO `exit` are OK.

The jump to the label always happens - it is not conditional.

Don't use GOTOs instead of DO...UNTIL or WHILE...ENDWH, as they make procedures difficult to understand.

▮ *Vectoring to a label: VECTOR/ENDV*

VECTOR jumps to one of a list of labels, according to the value in an integer variable. The list is terminated by the ENDV statement. For example:

```
VECTOR P%
  FUNCA,FUNCX
  FUNCR
ENDV
PRINT "P% was not 1/2/3" :GET :STOP
FUNCA::
PRINT "P% was 1" :GET :STOP
FUNCX::
PRINT "P% was 2" :GET :STOP
FUNCR::
PRINT "P% was 3" :GET :STOP
```

Here, if P% is 1, VECTOR jumps to the label FUNCA::. If it is 2, it jumps to FUNCX::, and if 3, FUNCR::. If P% is any other value, the program continues with the statement after the ENDV statement.

The list of labels may spread over several lines, as in this example, with a comma separating labels in any one line but no comma at the end of each line. Again, you can write each label in the list with a double colon, if you like.

VECTOR...ENDV can save you from having to write very long IF...ENDIF structures, with ELSEIF used many times.

Stopping a running program

This example introduces the STOP command. This stops a running program completely, just as if the end of the program had been reached. STOP has the same effect in a single procedure as using GOTO to jump to a label above the final ENDP.

★ *UNTIL 0, WHILE 1*

Zero and non-zero are logical values meaning 'false' and 'true' respectively. `UNTIL 0` and `WHILE 1` therefore mean 'do forever', since the condition 0 is never 'True' and the condition 1 is always 'True'. Use loops with these conditions when you need to check the real condition somewhere in the middle of the loop. When the real condition is met, you can BREAK out of the loop.

For example:

```
PROC test:
   WHILE 1
      ... REM some other lines here
      IF KEY :BREAK :ENDIF
      ... REM some other lines here
   ENDWH
ENDP
```

This example uses the KEY command. KEY returns 0 if no key has been pressed. When a key is pressed, KEY returns a non-zero value which counts as 'True', and the BREAK is executed.

★ SUMMARY

```
DO
   statements
UNTIL condition

WHILE condition
   statements
ENDWH

IF condition
   statements
ELSEIF condition
   statements
ELSE
   statements
ENDIF

VECTOR int%
   label1, label2
   label3...
ENDV
```

GOTO `label` jumps to `label::`

BREAK goes to the first line after the end of the loop - the line following the UNTIL or ENDWH line.

CONTINUE goes to the test condition of the loop - the UNTIL or the WHILE line.

STOP stops a running program completely.

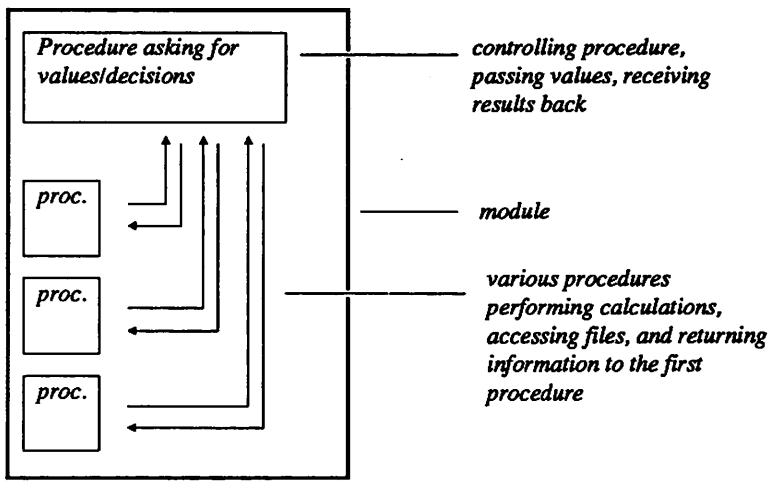
4

Calling procedures

Using more than one procedure

If you wanted a single procedure to perform a complex task, the procedure would become long and complicated. It is more convenient to have a module containing a number of procedures, each of which you can write and edit separately.

Many OPL modules are in fact a set of procedures linked up - each procedure doing just one job (such as a certain calculation) and then passing its results on to other procedures so they can do other operations:


Procedure asking for values/decisions
controlling procedure, passing values, receiving results back
proc.
proc.
proc.
module
various procedures performing calculations, accessing files, and returning information to the first procedure

OPL is designed to encourage programs written in this way, since:

- You can store all the procedures which make up a program in the same module file

and

- One procedure can *call*, that is run, another.

Modules containing more than one procedure

You can have as many procedures as you like in a module. Each must begin with PROC and end with ENDP.

When you run a translated module it is always the first procedure, at the top of the module, which is actually run. When this finishes, the module stops; any other procedures in the file are only run if and when they are called.

Although you can use any name you want, it's common to give the first procedure a name like `start`.

Procedures which run on their own should be written and translated as separate modules, otherwise you won't be able to run them.

Calling procedures

To run another procedure, simply give the name of the procedure (with the colon). For example, this module contains two procedures:

```
PROC one:
   PRINT "Start"
   PAUSE 40
   two: REM calls procedure two:
   PRINT "Finished"
   PAUSE 40
ENDP

PROC two:
   PRINT "Doing..."
   PAUSE 40
ENDP
```

Running this module would run procedure `one:`, with this effect: `Start` is displayed; after a PAUSE it calls `two:`, which displays `Doing...`; after another PAUSE `two:` *returns* to the `one:` procedure; `one:` displays `Finished`; and after a final PAUSE, `one:` finishes.

Calling procedures in other modules - LOADM

A procedure in one module can call procedures in another module, provided this module has been loaded with the LOADM command. LOADM needs the filename of the module to load.

For example, if an OPL module `MAIN` has these two procedures:

```
PROC main:
   LOADM "library"
   pthis: REM run pthis: in this module
   pother: REM run pother: in "library"
   PRINT "Finished"
   PAUSE 40
   UNLOADM "library"
ENDP
```

```
PROC pthis:
   PRINT "Running pthis:"
   PRINT "in this module"
   PAUSE 40
ENDP
```

and a module called LIBRARY has this one:

```
PROC pother:
   PRINT "Running pother:"
   PRINT "in module called LIBRARY"
   PAUSE 40
ENDP
```

then running MAIN would display `Running pthis:` and `in this module`; then display `Running pother:` and `in module called LIBRARY`; and then display `Finished`.

You can use LOADM up to three times, to load up to three modules. If you want to load a fourth module, you must first unload one of those currently loaded, by usingUNLOADM with the filename. To keep your program tidy, you should UNLOADM a module as soon as you have finished using it.

Uses of calling procedures

Calling procedures can be used to:

- Structure your programs more clearly so they're easier to adapt after you've written them
- Use the same procedure in different programs - say, to perform a certain common calculation.

For example, when your program asks you "Do this or do that?", make two procedure calls - either `this:` or `that:` procedure - depending on what you reply, for example:

```
PROC input:
   LOCAL a$(1)
   PRINT "Add [A] or Subtract [S]?:",
   a$=UPPER$(GET$)
   IF a$="A"
      add: REM first procedure
   ELSEIF a$="S"
      subtract: REM second procedure
   ENDIF
ENDP
```

To make full use of procedure calls, you must be able to communicate values between one procedure and another. There are two ways of doing this: *global variables* and *parameters*.

Parameters

Values can be passed from one procedure to another by using *parameters*. They look, and act, very much like arguments to functions.

In the example below, the procedure `price:` calls the procedure `tax:`. At the same time as it calls it, it passes a value (in this case, the value which INPUT gave to the variable `x`) to the parameter `p` named in the first line of `tax:`. The parameter `p` is rather like a new local variable inside `tax:`, and it has the value passed when `tax:` is called. (The `tax:` procedure is **not** changing the variable `x`.)

The `tax:` procedure displays the value of `x` plus 17.5% tax.

```
PROC price:
   LOCAL x
   PRINT "ENTER PRICE",
   INPUT x
   tax:(x) REM Passes the value of x to p
   GET
ENDP

PROC tax:(p)
   PRINT "PRICE INCLUDING TAX =",p*1.175
ENDP
```

- In the *called* procedure, follow the procedure name by the names to use for the parameters, in brackets and separated by commas - for example `proc2:(cost,profit)`.

The parameter type is specified as with variables - for example `p` for a floating-point parameter, `p%` for an integer, `p&` for a long integer, `p$` for a string. You can't have array parameters.

- In the *calling* procedure, the *values* for the parameters are given in brackets, in the right order and separated by commas, after the colon of the called procedure - for example `proc2:(60,30)`.

The values passed to the parameters may be the values of variables, strings in quotes, or constants. So a call might be `calc:(a$,x%,15.8)` and the first line of the called procedure `PROC calc:(name$,age%,salary)`

In the called procedure, you cannot assign values to parameters - for example, by the statement `p=10`.

You will see a 'Type mismatch' error displayed if you try to pass the wrong type of value to a parameter - for example, 45 to (a$).

Multiple parameters

In the following example, the second procedure `tax2:` has two parameters:

- The value of the price variable `x` is passed to the parameter `p1`.
- The value of the tax rate variable `r` is passed to the parameter `p2`.

`tax2:` displays the price plus tax at the rate specified.

```
PROC price2:
   LOCAL x,r
   PRINT "ENTER PRICE",
   INPUT x
   PRINT "ENTER TAX RATE",
   INPUT r
   tax2:(x,r)
   GET
ENDP

PROC tax2:(p1,p2)
   PRINT p1+p2 %
ENDP
```

This uses the `%` symbol as an operator - `p1+p2 %` means `p1` plus `p2` percent of `p1`. Note the space before the `%`; without it, `p2%` would be taken as representing an integer variable.

Appendix B has more about the `%` operator.

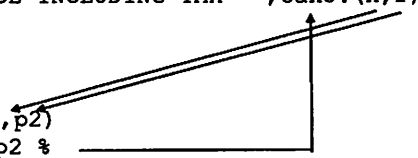
Returning values

In the following example, the RETURN command is used to return the value of `x` plus tax at `r` percent - to be displayed in `price3:`. This is very similar to the way functions return a value.

The `tax3:` procedure calculates, but doesn't display the result. This means it can be called by other procedures which need to perform this calculation but do not necessarily need to display it.

```
PROC price3:
   LOCAL x,r
   PRINT "ENTER PRICE",
   INPUT x
   PRINT "ENTER TAX RATE",
   INPUT r
   PRINT "PRICE INCLUDING TAX =",tax3:(x,r)
   GET
ENDP


PROC tax3:(p1,p2)
   RETURN p1+p2 %
ENDP
```



Only one value may be returned by the RETURN command.

The name of a procedure which returns a value must end with the correct identifier - $ for string, % for integer, or & for long integer. To return a floating-point number, it should end with none of these symbols. For example, PROC abcd$: can return a string, while PROC counter%: can return an integer. In this example, ref$: returns a string:

```
PROC refname:
   LOCAL a$(30),b$(2)
   PRINT "Enter reference and name:",
   INPUT a$
   b$=ref$:(a$)
   PRINT "Ref is:",b$
   GET
ENDP


PROC ref$:(name$)
   RETURN LEFT$(name$,2)
   REM LEFT$ takes first 2 letters of name$
ENDP
```

If you don't use the RETURN command, a string procedure returns the null string (""). Other (numeric) types of procedure return zero.

GLOBAL variables

You can only return one value with the RETURN command. If you need to pass back more than one value, use GLOBAL variables.

Instead of declaring `LOCAL x%,name$(5)` declare `GLOBAL x%,name$(5)`. The difference is that:

- Local variables are valid only in the procedure in which they are declared.
- Global variables can also be used in any procedures (including those in loaded modules) called by the procedure in which they are declared.

So this module would run OK:

```
PROC one:
   GLOBAL a%
   PRINT a%
   two:
   GET
ENDP

PROC two:
   a%=2 REM Sees a% declared in one:
   PRINT a%
ENDP
```

When you run this, the value 0 is displayed first, and then the value 2.

You would see an 'Undefined externals' error displayed if you used LOCAL instead of GLOBAL to declare `a%`, since the procedure `two:` wouldn't recognise the variable `a%`. To avoid errors, though, use the LOCAL command unless you really need to use GLOBAL.

A local declaration overrides a global declaration in that procedure. So if `GLOBAL a%` was declared in a procedure, which called another procedure in which `LOCAL a%` was declared, any modifications to the value of `a%` in this procedure would not effect the value of the global variable `a%`.

Passing back values

You can effectively pass as many values as you like back from one procedure to another by using global variables. *Any modifications to the value of a variable in a called procedure are automatically registered in the calling procedure.*

For example:

```
PROC start:
   GLOBAL varone,vartwo
   varone=2.5
   vartwo=2
   op:
   PRINT varone,vartwo
   GET
ENDP


PROC op:
   varone=varone*2
   vartwo=vartwo*4
ENDP
```

This would display 5 8

Calling procedures by strings

Procedures can be called using a string expression for the procedure name. Use an `@` symbol, optionally followed by a character to show what type of value is returned - for example, `%` for an integer. Follow this with the string expression in brackets. You can use upper or lower case characters.

Here are examples showing the four types of value which can be returned:

`i% = @%(name$):(parameters)` for integer
`l& = @&(name$):(parameters)` for long integer
`s$ = @$(name$):(parameters)` for string
`f = @(name$):(parameters)` for floating-point

So, for example, `test$:(1.0,2)` and `@$("test"):(1.0,2)` are equivalent.

Note that the string expression does not itself include a symbol for the type (`%`, `&` or `$`).

You may find this useful as you start to build more complex programs. The chapter on 'Friendlier Interaction' includes an example program which uses this method.

★ *'Undefined externals' error*

If, perhaps because of a typing error, you use a name which is not one of your variables, no error occurs when you translate the module. This is because it could be the name of a global variable, declared in a different procedure, which might be available when the procedure in question was called. If no such global variable is available, an 'Undefined externals' error is shown. This also displays the variable name which caused the error, together with the module and procedure names, in this format: 'Error in MODULE\PROCEDURE,VARIABLE'.

★ *SUMMARY*

Call a procedure by stating its name, including the colon.

Pass parameters to a procedure by following the procedure call with the values for the parameters, e.g. `calc2:(4.5,32)`. In the called procedure, follow the procedure name with the parameter names, e.g. `PROC calc2:(mod,div%)`.

To make variables declared in one procedure accessible to called procedures, declare the variables with GLOBAL instead of LOCAL.

LOADM and UNLOADM load (make available) and unload other modules of procedures.

5

Data file handling

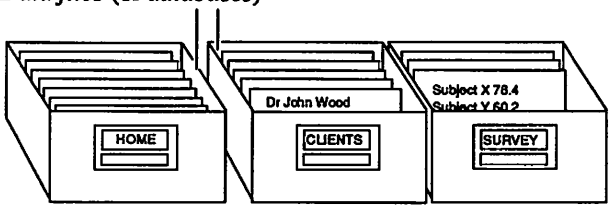
You can use OPL to create data files (databases) like those used by the Database. You can store any kind of information in a data file, and retrieve it for display, editing or calculations.

This chapter covers:

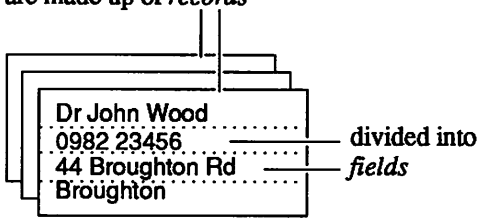
- **Creating data files**
- **Adding and editing records**
- **Searching records**
- **Using a data file both in OPL and in the Database**

Files, records and fields

Data files (or *databases*)



are made up of *records*



For example, in a data file of names and addresses, each record might have a name field, a telephone number field, and separate fields for each line of the address.

In OPL you can:

- Create a new ***file*** with CREATE, or open an existing file with OPEN, and copy, delete or rename a file with COPY, DELETE and RENAME.
- Add a new ***record*** with APPEND, change an existing one with UPDATE, and remove a record with ERASE.
- Fill in a ***field*** by assigning a value to a field variable.

Creating a data file

Use the CREATE command like this:

```
CREATE filename$,logical name,field1,field2,...
```

For example:

```
CREATE "clients",B,nm$,tel$,ad1$,ad2$,ad3$
```

creates a data file called CLIENTS.

The file name is a string, so remember to put quote marks around it. You can also assign the name string to a string variable (for example `fil$="clients"`) and then use the variable name as the parameter - `CREATE fil$,A,field1,field2`.

Logical names

You can have up to 4 data files open at a time. Each of these must have a logical name: A, B, C or D. The logical name lets you refer to this file without having to keep using the the full file name.

A different logical name must be used for each data file opened - one called A, one called B, one called C and one called D. A file does not have to be opened with the same logical name as the last time it was opened. When a file is closed, its logical name is freed for use by another file.

Fields

`field1, field2,...` are the field names - up to 32 in any record. These are like variables, so use `%` `&` or `$` to make the appropriate types of fields for your data. You cannot use arrays. Do not specify the maximum length of strings that the string fields can handle. The length is automatically set at 255 characters.

Field names may be up to 8 characters long, including any qualifier like `&`.

When referring to fields, add the logical file name to the front of the field name, to specify which opened file the fields belong to. Separate the two by a dot. For example, `A.name$` is the `name$` field of the file with logical name A, and `C.age%` is the `age%` field of the file with logical name C.

The values of all the fields are 0 or null to start with. You can see this if you run this example program:

```
PROC creatfil:
   CREATE "example",A,int%,long&,float,str$
   PRINT "integer=";a.int%
   PRINT "long=";a.long&
   PRINT "float=";a.float
   PRINT "string=";a.str$
   CLOSE
   GET
ENDP
```

Opening a file

When you first CREATE a data file it is automatically open, but it closes again when the program ends. **If a file already exists, trying to CREATE it again will give an error**- so if you ran the procedure `creatfil:` a second time you would get an error. To open an existing file, use the OPEN command.

OPEN works in the same way as the CREATE command. For example:

```
OPEN "clients",B,a$,b$,c$,d$,e$
```

- You must use the same filename as when you first created it.
- You must include in the OPEN command each of the fields you intend to alter or read. You can omit fields from the end of the list; you **cannot** miss one out from the middle of the list, for example `field1$,,name$` They must remain the same type of field, but you can change their names. So a file created with fields `name$,age%` could later be opened with the fields `a$,x%`.
- Give the file a logical name. Up to 4 files may be open at any one time, with logical names A, B, C and D. You can't have two files open simultaneously with the same logical name, so when opening the files, remember which logical names you have already used.

In a **different** module from the one containing `creatfil:` you could enter these two procedures:

```
PROC openfile:
   IF NOT EXIST("example")
      CREATE "example",A,int%,lng&,fp,str$
   ELSE
      OPEN "example",A,int%,lng&,fp,str$
   ENDIF
   PRINT "Current values:"
   show:
   PRINT "Assigning values"
   A.int%=1
   A.lng&=&2**20 REM avoids integer overflow
   A.fp=SIN(PI/6)
   PRINT "Give a value for the string:"
   INPUT A.str$
   PRINT "New values:"
   show:
   CLOSE
ENDP

PROC show:
   PRINT "integer=";A.int%
   PRINT "long=";A.lng&
   PRINT "float=";A.fp
   PRINT "string=";A.str$
   GET
ENDP
```

Notes

Opening/closing the file

The IF...ENDIF checks to see if the file already exists, using the EXIST function. If it does, the file is opened; if it doesn't, the file is created.

You should always 'close' a data file (with the CLOSE command) when you have finished using it. Data files close automatically when programs end. You can only have 4 files open at a time - if you have 4 files open and you want to access another one, close one of them. CLOSE closes the *current* file.

Giving values to the fields

The fields can be assigned values just like variables. The field name **must** be used with the logical file name like this: `A.f%=1` or `INPUT A.f$`.

If you try to give the wrong type of value to a field (for example "Davies" to `f%`) an error message will be displayed.

You can access the fields from other procedures, just like global variables. Here the called procedure `show:` displays the values of the fields.

Field names
You must know the type of each field, and you must give each a separate name - you cannot refer to the fields in any indexed way, e.g. as an array.

Opening a file for sharing
The OPENR command works in exactly the same way as OPEN, except that the file cannot be written to (with UPDATE or APPEND), only read. However, more than one running program can then look at the file at the same time.

Saving records

The last example procedure did not actually save the field values as a record to a file. To do this you need to use the APPEND command. This program, for example, allows you to add records to the `example` data file:

```
PROC count:
   LOCAL reply%
   OPEN "example",A,f%,f&,f,f$
   DO
      CLS
      AT 20,1 :PRINT "Record count=";COUNT
      AT 9,5 :PRINT "(A)dd a record"
      AT 9,7 :PRINT "(Q)uit"
      reply%=GET
      IF reply%=%q OR reply%=%Q
         BREAK
      ELSEIF reply%=%A OR reply%=%a
         add:
      ELSE
         BEEP 16,250
      ENDIF
   UNTIL 0
   CLOSE
ENDP

PROC add:
   CLS
   PRINT "Enter integer field:";
   INPUT A.f%
   PRINT "Enter long integer field:";
   INPUT A.f&
   PRINT "Enter numeric field:";
   INPUT A.f
   PRINT "Enter string field:";
   INPUT A.f$
   APPEND
ENDP
```

BEEP
The BEEP command makes a beep of varying pitch and length:

```
BEEP duration%,pitch%.
```

The duration is measured in 1/32s of a second, so `duration%=32` would give a beep a second long. Try `pitch%=500` for a high beep, or `50` for a low beep.

The number of records

The COUNT function returns the number of records in the file. If you use it just after creating a database, it will return 0. As you add records the count increases.

How the values are saved

Use the APPEND command. This has no parameters. The values assigned to `A.f%`, `A.f&`, `A.f` and `A.f$` are added as a new record to the end of the EXAMPLE data file. If you only give values to some of the fields, not all, you won't see any error message; null strings ("") will be given to string fields, and zero to numeric fields when you APPEND.

New field values are always added to the end of the current data file - as the last record in the file (if the file is a new one, it will also be the first record).

At any time while a data file is open, the field names currently in use can be used like any other variable - for example, in a PRINT statement, or a string or numeric expression.

APPEND and UPDATE
APPEND adds the current field values to the end of the file as a new record, whereas UPDATE deletes the **current** record and then adds the current field values to the end of the file as a new record.

Changing the current file

Immediately after a file has been created or opened, it is automatically current. This means that the APPEND or UPDATE commands save records to this file, and the record-position commands (explained below) move around this file. You can still use the fields of other open files, for example `A.field1=B.field2`

USE makes current one of the other opened files. For example `USE B` selects the file with the logical name B (as specified in the OPEN or CREATE command which opened it).

If you attempt to USE a file which has not yet been opened or created, an error is reported.

In this procedure, the EOF function checks whether you are at the end of the current data file - that is, whether you've gone past the last record. You can use EOF in the test condition of a loop - UNTIL EOF or WHILE NOT EOF in order to carry out a set of actions on all the records in a file.

Example - copies selected records from one file to another

```
PROC copyrec:
   OPEN "example",A,f%,f&,f,f$
   TRAP DELETE "temp"
   REM If file doesn't exist, ignore error
   CREATE "temp",B,f%,f&,f,f$
   PRINT "Copying EXAMPLE to TEMP"
   USE A REM the EXAMPLE file
   DO
      IF a.f%>30 and a.f<3.1415
         b.f%=a.f%
         b.f&=a.f&
         b.f=a.f
         b.f$="Selective copy"
         USE B REM the TEMP file
         APPEND
         USE A
      ENDIF
      NEXT
   UNTIL EOF REM until End Of File
   CLOSE REM closes A; B becomes current!
   CLOSE REM closes B
ENDP
```

This example uses the DELETE command to delete any `temp` file whch may exist, before making it afresh. Normally, if there was no `temp` file and you tried to delete it, an error would be generated. However, this example uses TRAP with the DELETE command. TRAP followed by a command means "if an error occurs in the command, carry on regardless".

There are more details of TRAP in the chapter on 'Error Handling'.

Moving from record to record

When you open or create a file, the first record in the file is current. To read, edit, or erase another record, you must make that record current - that is, move to it. Only one record is current at a time. To change the current record, use one of these commands:

POSITION 'moves to' a particular record, setting the field variables to the values in that record. For example, the instruction `POSITION 3` makes record 3 the current record. The first record is record 1.

You can find the current record number by using the POS function, which returns the number of the current record.

FIRST moves to the first record in a file.

NEXT moves to the following record in a file. If the end of the file is passed, NEXT does not report an error, but the current record is a new, empty record. This case can be tested for with the EOF function.

BACK moves to the previous record in the file. If the current record is the first record in the file then that first record stays current.

LAST moves to the last record in the file.

Deleting a record

ERASE deletes the current record in the current file.

The next record is then current. If the erased record was the last record in a file, then following this command the current record will be empty and EOF will return true.

Finding a record

FIND makes current the next record which has a field matching your search string. Capitals and lower-case letters match. For example:

```
r%=FIND("Brown")
```

would select the first record containing a string field with the value "Brown", "brown" or "BROWN", etc. The number of that record is returned, in this case to the variable `r%`. If the number returned is zero, no matching field was found. Any other number means that a match was found.

The search includes the current record. So after finding a matching record, you need to use NEXT before you can continue searching through the following records.

`FIND("Brown")` would not find a field "Mr Brown". To find this, use wildcards, as explained below.

You can only search string fields, not number fields. For example, if you assigned the value 71 to the field `a%`, you could not find this with FIND. But if you assigned the value "71" to `a$`, you could find this.

Wildcards

`r%=FIND("*Brown*")` would make current the next record containing a string field in which `Brown` occurred - for example, the fields MR BROWN, `Brown A.R.` and `Browns Plumbing` would be matched. The wildcards you can use are:

? matches any one character

* matches any number of characters.

Once you've found a matching record, you might display it on the screen, erase it or edit it. For example, to display all the records containing "BROWN":

```
FIRST
WHILE FIND("*BROWN*")
   PRINT a.name$, a.phone$
   NEXT
   GET
ENDWH
```

Data files and the Database

The files you use with the Database (listed under the Data icon in the System screen) are also data files. Data files created in OPL can be viewed by the Database, and vice versa.

In OPL: to open a data file made by the Database, begin its name with `\DAT\`, and end it with `.DBF`. For example, to open the file called `data` which the Database normally uses:

```
OPEN "\dat\data.dbf",A,a$,b$,c$,d$...
```

Restrictions:

- You can use up to 32 field variables, all strings. It is possible for records to contain more than 32 fields, but these fields cannot be accessed by OPL. It's safe to change such a record and use UPDATE, as the extra fields will remain unchanged.
- The maximum record length in OPL is 1022 characters. You will see a 'Record too large' error (-43) if your program tries to open a file which contains a record longer than this.
- The Database breaks up long records (over 255 characters) when storing them. They would appear as separate records to OPL.

In the Database: to examine an OPL data file, press the Data button, select 'Open' from the 'File' menu, and type the name with `\OPD\` on the front and `.ODB` at the end - for example:

```
\opd\example.odb
```

Restrictions:

- All of the fields must be string fields.
- You can have up to a maximum of 32 fields, as specified in the CREATE command. If you view an OPL data file with the Database, and add more lines to records than the number of fields specified in the original CREATE command, you will get an error if you subsequently try to access these additional fields in OPL.

In both cases, you are using a more complete *file specification*. There is more about file specifications in the Advanced Topics chapter.

6

Graphics

OPL graphics allows you, for example, to:

- **Draw lines and boxes.**
- **Fill areas with patterns.**
- **Display text in a variety of styles, at any position on the screen.**
- **Scroll areas of the screen.**
- **Manipulate windows and bit patterns.**
- **Read data back from the screen.**

Graphics keywords begin a `G`**. In this manual a lower case** `g` **is used - for example,** `gBOX` **- to distinguish them from other commands such as** `GET` **and** `GOTO`**. As with other OPL commands, though, you can type them using upper or lower case letters, so** `GBOX`**,** `gbox` **and** `gBOX` **all mean the same thing.**

Simple graphics

The Series 3 screen is made up of a regular pattern of 240 points across by 80 down. These points are sometimes known as *pixels*. If you increase the screen contrast for a moment, by holding the PSION key and pressing >, the pattern of pixels will stand out.

Each pixel is identified by two numbers, giving its position across and down from the top left corner of the screen. 0,0 denotes the pixel in the top left corner; 2,1 is the pixel 2 points across and one down, and so on. 239,79 is the pixel in the bottom right corner.

Note that these co-ordinates are very different to the cursor positions set by the AT command.

OPL maintains a *current position* on the screen. Graphics commands which draw on the screen generally take effect at this position. Initially, the current position is the top left corner, 0,0.

Screen positions

Drawing lines

Here is a simple procedure to draw a horizontal line in the middle of the screen:

```
PROC lines:
   gMOVE 90,40
   gLINEBY 60,0
   GET
ENDP
```

gMOVE moves the current position by the specified amount. In this case, it moves the current position 90 pixels right and 40 down, from 0,0 to 90,40. It does not display anything on the screen.

gLINEBY (g-Line-By) draws a line from the current position (just set to 90,40) to a point at the distance you specify - in this case to a point 60 to the right and 0 down, i.e. 150,40. gLINEBY also has the effect of moving the current position to the end of the line it draws.

With both gMOVE and gLINEBY, you specify positions **relative to the current position.** Most OPL graphics commands do likewise. gMOVE and gLINEBY, however, do have corresponding commands which use absolute

pixel positions. gAT moves to the pixel position you specify; gLINETO draws a line from the current position to an absolute position. The horizontal line procedure could instead be written:

```
PROC lines:
   gAT 90,40
   gLINETO 150,40
   GET
ENDP
```

gAT and gLINETO may be useful in very short graphics programs, and gAT is always the obvious command for moving to a particular point on the screen, before you start drawing. But once you do start drawing, use gMOVE and gLINEBY. They make it much easier to develop and change programs, and allow you to make useful graphics procedures which can display things anywhere you set the current position. Almost all graphics drawing commands use relative positioning for these reasons.

Drawing dots

You can set the pixel at the current position with `gLINEBY 0,0`.

Right and down, left and up

gMOVE and gLINEBY find the position to use by adding the numbers you specify onto the current position. If the numbers are positive, it moves to the right and down the screen. If you use negative numbers, however, you can specify positions to the left of and/or above the current position. For example, this procedure draws the same horizontal line as before, then another one above it:

```
PROC lines2:
   gMOVE 90,40
   gLINEBY 60,0
   gMOVE 0,-10
   gLINEBY -60,0
   GET
ENDP
```

The first two program lines are the same as before. gLINEBY moves the current position to the end of the line it draws, so after the first gLINEBY the current position is 150,40. The second gMOVE moves the current position **up** by 10 pixels; the second gLINEBY draws a line to a point 60 pixels to the **left**.

▮ *Going off the screen*

No error is reported if you try to draw off the edge of the screen. It is quite possible to leave the current position off the screen - for example, `gLINETO 300,40` will draw a line from the current position to some point on the right hand screen edge, but the current position will finish as 300,40.

There's no harm in the current position being off the screen. It allows you to write procedures which display a particular pattern at the current position, and not have to worry whether that position is too close to the edge of the screen for the whole pattern to fit on.

Overwriting pixels

▮ *Drawing rectangles*

The gBOX command draws a box outline. For example, `gBOX 50,10` draws a box from the current position to a point 50 pixels to the right and 10 down. If the current position were 100,20, the four corners of this box would be at 100,20, 150,20, 150,30 and 100,30.

gBOX does not change the current position.

gFILL draws a filled box in the same way as gBOX draws a box outline, but it has a third parameter to say which pixels to set. If it set to 0, the pixels which make up the box would be set. If set to 1, pixels are cleared; if set to 2, they are inverted, that is, pixels already set on the screen become cleared, and vice versa. The values 1 and 2 are used when **overwriting** areas of the screen which already have pixels set.

For example, this procedure displays a "robot" face, using gFILL to draw set and cleared boxes:

```
PROC face:
   gFILL 60,60,0 REM set the entire face
   gMOVE 5,10 :gFILL 15,10,1 REM left eye
   gMOVE 35,0 :gFILL 15,10,1 REM right eye
   gMOVE -15,15 :gFILL 10,15,1 REM nose
   gMOVE -10,20 :gFILL 30,10,1 REM mouth
   GET
ENDP
```

Before calling such a procedure, you would set the current position to be where you wanted the top left corner of the head.

You could make the robot wink with the following procedure, which inverts part of one eye:

```
PROC wink:
   gMOVE 5,10 REM move to left eye
   gFILL 15,7,2 REM invert most of the eye
   PAUSE 10
   gFILL 15,7,2 REM invert it back again
   GET
ENDP
```

Again, you would set the current position before calling this.

The gPATT command can be used to draw a shaded filled rectangle. To do this, use -1 as its first parameter, then the same three parameters as for gFILL - width, height, and overwrite method. Overwrite methods 0, 1 and 2 apply only to the pixels which are 'on' in the shading pattern. Whatever was on the screen may still show through, as those pixels which are 'clear' in the shading pattern are left as they were.

To completely overwrite what was on the screen with the shaded pattern, gPATT has an extra overwrite method of 3. So, for example, `gPATT -1,60,60,3` in the first procedure would have displayed a shaded robot head, whatever may have been on the screen.

▮ *Overwriting with any drawing command*

By using the gGMODE command, any drawing command such as gLINEBY or gBOX can be made to clear or invert pixels, instead of setting them. gGMODE sets the effect of **all** subsequent drawing commands.

The values are the same as for gFILL: `gGMODE 1` for clearing pixels, `gGMODE 2` for inverting pixels, and `gGMODE 0` to return to setting pixels.

For example, some white lines can give the robot a furrowed brow:

```
PROC brow:
   gGMODE 1 REM gLINEBY will now clear pixels
   gMOVE 5,4 :gLINEBY 50,0
   gMOVE 0,2 :gLINEBY -50,0
   gGMODE 0
   GET
ENDP
```

▮ *Other drawing keywords*

- gPOLY: draw a sequence of lines.
- gINVERT: invert a rectangular area, except for its four corner pixels.

See the alphabetic listing for more details of these keywords.

Graphical text

Displaying text with gPRINT

The PRINT command displays text in one standard font, in a screen area controlled by the SCREEN command. You can, however, display text in a variety of fonts and styles, at any pixel position, with gPRINT.

gPRINT is a graphical version of PRINT, and displays a list of expressions in a similar way. Some examples:

```
gPRINT "Hello",name$
gPRINT a$
gPRINT "Sin(PI/3) is",sin(pi/3)
```

Unlike PRINT, gPRINT does not end by moving to a new line. A comma between expressions is still displayed as a space, but a semi-colon has no effect. gPRINT used on its own does nothing.

The first character displayed has its left side and baseline at the current position. The *baseline* is like a line on lined notepaper - graphically, this is the horizontal line which includes the lowest pixels of upper case characters. Note that some characters, such as 'g', 'j', 'p', 'q' and 'y', set pixels below the baseline.

While CURSOR ON displays a flashing cursor for ordinary text displayed with PRINT, CURSOR 1 switches on a cursor for graphical text. CURSOR OFF removes either cursor.

Fonts

The gFONT command sets the font to be used by subsequent gPRINT commands.

gFONT 1 sets the standard font, which gPRINT normally uses. gFONT 2 sets the bold font. (gFONT 3 gives the small digit font, which can **only** show the the numbers 1 to 9, and a space.)

This procedure shows you examples of the fonts:

```
PROC exfont:
   gAT 20,20 :gFONT 1
   gPRINT "Normal font looks like this"
   gAT 20,40 :gFONT 2
   gPRINT "Bold font looks like this"
   gAT 20,60 :gFONT 3
   gPRINT "1 2 3 4 5 6 7 8 9 0"
   GET
ENDP
```

▮ *Text style*

The gSTYLE command sets the text style to be used by subsequent gPRINT commands.

Choose from these styles:

```
gSTYLE 1    bold
gSTYLE 2    underlined
gSTYLE 4    inverse
gSTYLE 8    double height
gSTYLE 16   mono
gSTYLE 32   italic
```

The 'mono' style is not proportionally spaced - each character is displayed with the same width. This is the way PRINT displays characters.

You can combine these styles by adding the relevant numbers together. `gSTYLE 12` sets the text style to inverse and double-height (4+8=12). Here's an example of this style:

```
PROC style:
   gAT 20,50 :gFONT 1
   gSTYLE 12 :gPRINT "Attention!"
   GET
ENDP
```

Use `gSTYLE 0` to reset to normal style.

The bold style provides a way to make your own fonts appear bolded - see gLOADFONT for details of user-defined fonts. For ordinary bold text, it's best to use `gFONT 1`.

▮ *Overwriting with gPRINT*

gPRINT normally displays text as if writing it with a black pen - the pixels that make up each letter are set, and that is all. If you're using areas of the screen which already have pixels set, or are even completely black, use gTMODE to change the way gPRINT displays the text.

gTMODE controls the display of text in the same way as gGMODE controls the display of lines and boxes. The values you use with gTMODE are similar to those for gGMODE: `gTMODE 1` for clearing pixels, `gTMODE 2` for inverting pixels, and `gTMODE 0` to return to setting pixels. There is also `gTMODE 3` which sets the pixels of each character while clearing the character's background. This is very useful as it guarantees that the text is readable.

This procedure shows the various effects possible via gTMODE:

```
PROC tmode:
   gAT 80,10 :gFILL 80,60,0 REM Black box
   gAT 110,10 :gFILL 20,60,1 REM White box
   gFONT 1 :gSTYLE 0
   gAT 90,20 :gTMODE 0 :gPRINT "ABCDEFGHIJK"
   gAT 90,30 :gTMODE 1 :gPRINT "ABCDEFGHIJK"
   gAT 90,40 :gTMODE 2 :gPRINT "ABCDEFGHIJK"
   gAT 90,50 :gTMODE 3 :gPRINT "ABCDEFGHIJK"
   GET
ENDP
```

Other graphical text keywords

- gPRINTB:Display text left aligned, right aligned or centred, in a cleared box.
- gXPRINT: Display text underlined/highlighted.
- gPRINTCLIP: Display text clipped to whole characters.
- gTWIDTH:Find width required by text.

All of these keywords take the current font and style into account, and work on a single string. See the alphabetic listing for more details.

Windows

So far, you've used the whole of the screen for displaying graphics. You can, however, use *windows* - rectangular areas of the screen.

OPL allows a program to use up to eight windows at any one time.

For OPL graphics keywords, **positions apply to the window you are using at any given time.** The point 0,0 means the top left corner of the current window, **not** the top left corner of the screen.

Window IDs and the default window

Each window has an ID number, allowing you to specify which window you want to work with at any time.

When a program first runs, it has one window called the *default window*. Its ID is 1, it is the full size of the screen, and initially all graphics commands operate on this window.

Other windows you create will have IDs from 2 to 8. **When you make another window the *current window*, all subsequent graphics commands operate on it.**

The first half of this chapter used only the default window. However, everything actually applies to the current window. For example, if you make a small window current and try to draw a very long line, the current position moves off past the window edge, and only that part of the line which fits in the window is displayed.

Creating new windows

The gCREATE function sets up a new window on the screen. It returns an ID number for the window. Whenever you want to change to this window, use gUSE with this ID.

Here is an example using gCREATE and gUSE, showing the point 20,20 in both the default window and a created window.

```
PROC windows:
   LOCAL id%
   id%=gCREATE(60,40,140,30,1)
   gBORDER 0 :gAT 20,20 :gLINEBY 0,0
   gPRINT " 20,20 (new)"
   GET
   gUSE 1 :gAT 20,20 :gLINEBY 0,0
   gPRINT " 20,20 (default)"
   GET
   gUSE id%
   gPRINT " Back."
   GET
ENDP
```

The line `id%=gCREATE(60,40,140,30,1)` creates a window with its top left corner at 60,40 on the screen. The window is set to be 140 pixels wide and 30 pixels deep. (You can use any integer values for these parameters, even if it creates the window partially or totally off the screen.) The fifth parameter to gCREATE specifies whether the window should immediately be visible or not; 0 means invisible, 1 (as here) means visible.

gCREATE makes the created window the current window, and sets the current position in it to 0,0. It returns an ID number for this window, which in this case is saved in the variable `id%`.

The `gBORDER 0` command draws a border one pixel wide around the current window. Here this is to show the position and size of the window. (gBORDER can draw a variety of borders - see the alphabetic listing for more details.)

The program then sets the pixel at 20,20 in this new window.

`gUSE 1` sets OPL back to the default window. The program then shows 20,20 in this window.

Finally, `gUSE id%` sets OPL to use the created window again, and a final message is displayed. This shows two more things:

- Each window has its own current position. The current position in the created window was remembered while the program went back to the default window.
- While using the created window, you cannot display anything outside its limits.

Closing windows

When you've finished with a particular window, close it with gCLOSE followed by its ID - for example, `gCLOSE 2`. You can create and close as many windows as you like, as long as there are only eight or fewer at any one time.

If you close the current window, the default window (ID=1) becomes current.

An error is raised if you try to close the default window.

When windows overlap

Windows can overlap on the screen, or even hide each other entirely. Use the gORDER command to control the foreground/background positions of overlapping windows.

`gORDER 3,1` sets the window whose ID is 3 to be in the foreground. This guarantees that it will be wholly visible. `gORDER 3 2` makes it second in the list; unless the foreground window overlaps it, it too will be visible.

Any position greater than the number of windows you have is interpreted as the end of the list. `gORDER 3,9` will therefore always force this window to the background, behind all others.

Note in particular that making a window the current window does not bring it to the foreground. You can make a background window current and display all kinds of things to it, but nothing will happen on the screen until you bring it to the foreground with gORDER.

When a window is first created with gCREATE it always becomes the foreground window, as well as the current window.

Hiding windows

If you are going to use several drawing commands on a particular window, you may like to make it invisible while doing so. When you then make it visible again, having completed the drawing commands, the whole pattern appears on the screen in one go, instead of being built up slowly.

Use `gVISIBLE ON` and `gVISIBLE OFF` to perform this function on the current window. You can also make new windows invisible as you create them, by using 0 as the fifth parameter to the gCREATE command, and you can hide windows behind other windows.

The graphics cursor in windows

To make the graphics cursor appear in a particular window, use the CURSOR command with the ID of the window. It will appear flashing at the current position in that window, provided it is not obscured by some other window.

The window you specify does not have to be the current window, and does not become current; you can have the cursor in one window while displaying graphical text in another. If you want to move to different window and put the graphics cursor in it, you must use both gUSE and CURSOR.

Since the default window always has an ID of 1, `CURSOR 1` will, as mentioned earlier, put the graphics cursor in it.

CURSOR OFF turns off the cursor, wherever it is.

Information about your windows

You don't have to keep a complete copy of all the information pertaining to each window you use. These functions return information about the current window:

- gIDENTITY returns its ID number.
- gRANK returns its foreground/background position, from 1 to 8.
- gWIDTH and gHEIGHT return its size.
- gORIGINX and gORIGINY return its screen position.
- gINFO returns information about the font, style and overwrite modes in use.
- gX and gY return the current position.

See the alphabetic listing for more details.

Other window keywords

- gCLS clears the current window.
- gSETWIN changes the position, and optionally the size, of the current window. (You **can** use this command on the default window, if you wish, but you **must** also use the SCREEN command to ensure that the area for PRINT commands to use is wholly contained within the default window.)
- gSCROLL scrolls all or part of the current window.

- gPATT fills an area in the current window with repetitions of another window, or with a grey pattern.
- gCOPY copies an area from another window into the current window.
- gSAVEBIT saves part or all of a window as a *bitmap file*.
- gPEEKLINE reads back a horizontal line of data from a specified window.

See the alphabetic listing for more details.

Advanced graphics

This section should provide a taste of some of the more exotic things you can do with OPL graphics. See the alphabetic listing for more details of the commands mentioned.

Bitmaps

A *bitmap* is an area in memory which acts just like an off-screen window. You can create bitmaps with gCREATEBIT. They have the following uses:

- You can manipulate an image in a bitmap before copying it with gPATT or gCOPY to a window on the screen. This is generally faster than manipulating an image in a hidden window.
- You can load *bitmap files* into bitmaps in memory, then copy them to on-screen windows.

OPL treats a bitmap as the equivalent of a window:

- Both are identified by ID numbers. Only one window or bitmap is current at any one time, set by gUSE.
- If you use bitmaps as well as windows, the **total** number must be eight or fewer.
- The top left corner of the current bitmap is still referred to as 0,0, even though it is not on the screen at all.

Together, windows and bitmaps are known as *drawables* - places you can draw to.

The following commands can be used with bitmaps in the same way as with windows: gUSE, gCLOSE, gCLS, gPATT, gCOPY gPEEKLINE, gSAVEBIT, gSCROLL, gIDENTITY gWIDTH, gHEIGHT and gINFO. These commands are described earlier in this chapter.

Use gLOADBIT to load a bitmap file into a bitmap in memory. You can save part or all of a window or bitmap to a bitmap file, with gSAVEBIT.

Speed improvements

The Series 3 screen is usually updated whenever you display anything on it. gUPDATE OFF switches off this feature. The screen will be updated as few times as possible, although you can force an update by using the gUPDATE command on its own.

This can result in a considerable speed improvement in some cases. You might, for example, use `gUPDATE OFF`, then a sequence of graphics commands, followed by `gUPDATE`. You should certainly use gUPDATE OFF if you are about to write exclusively to bitmaps.

gUPDATE ON returns to normal screen updating.

The gPOLY command draws a sequence of lines, as if by gLINEBY and gMOVE commands. If you have to draw a lot of lines (or dots, with `gLINEBY 0,0`), gPOLY can greatly reduce the time taken to do so.

Displaying a running clock

gCLOCK displays or removes a clock showing the system time. The clock can be digital or conventional.

User-defined fonts and cursor

If you have a user-defined font you can load it into memory with gLOADFONT. This returns an ID for the font; use this with gFONT to make the font current. The gUNLOADFONT command removes a user-defined font from memory when you have finished using it.

You can use three extra parameters with the CURSOR command to specify the ascent, width and height of the cursor. The *ascent* is the number of pixels (-128 to 127) by which the top of the cursor should be above the baseline of the current font. The height and width parameters should both be between 0 and 255. For example, `CURSOR 1,12,4,14` sets a cursor 4 pixels wide by 14 high in the default window (ID=1), with the cursor top at 12 pixels above the font baseline.

If you do not use these parameters, the cursor is 2 pixels wide, and has the same height and ascent as the current font.

The gINFO command returns information about the cursor - see the alphabetic listing for more details.

The text and graphics windows

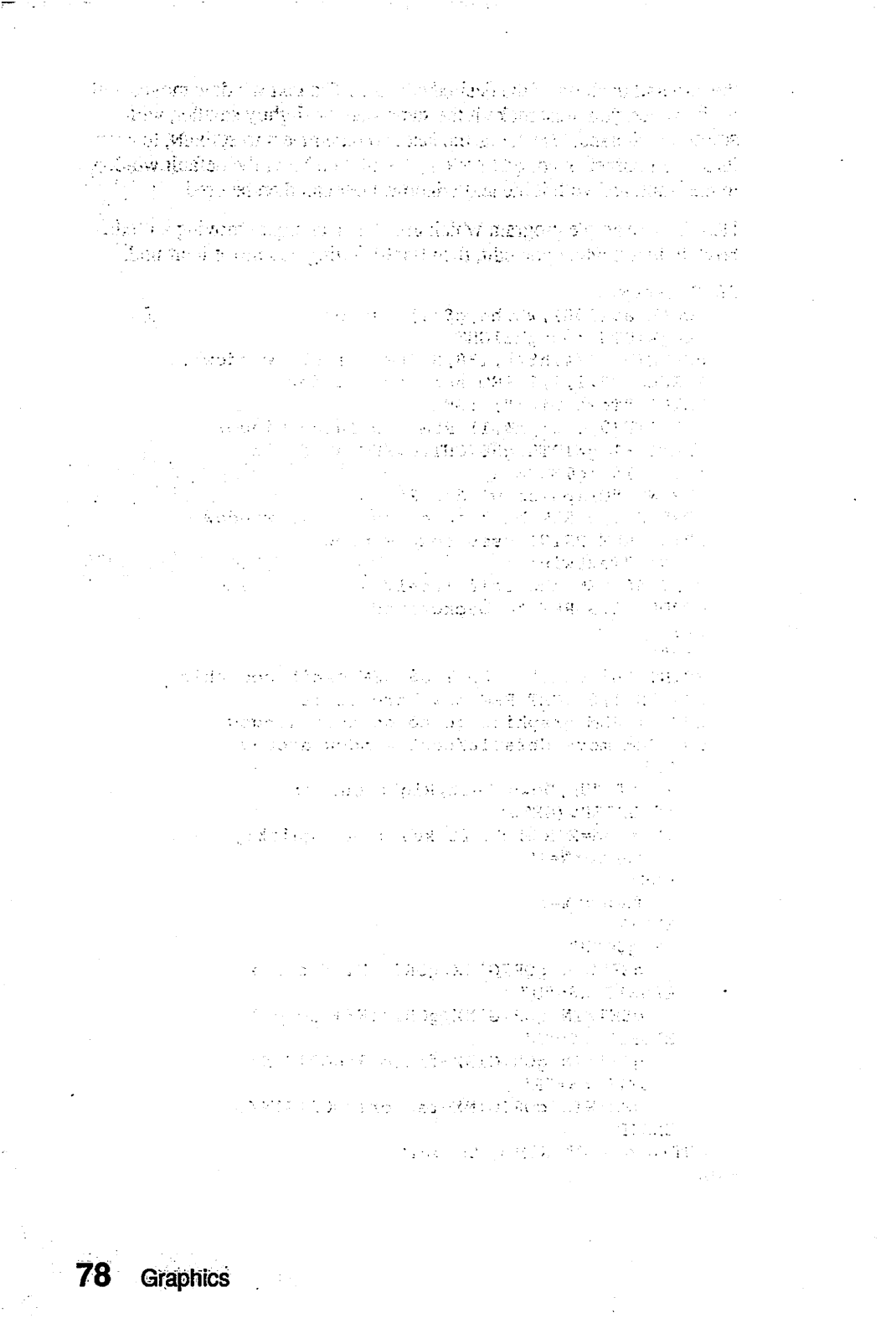
PRINT displays mono-spaced text in a *text window*. You can set the size of this with the SCREEN command. If you use graphics windows, they may hide any text you display with PRINT.

If you do need to use the text window, for example to use functions like EDIT, it's easiest to use SCREEN to place it out of the way of your graphics windows. You can, however, use it on top of a graphics window - for example, you might want to use EDIT to simulate an edit box in the graphics window. Use gSETWIN to change the **default window** to about

the size and position of the desired edit box. The text window moves with it - however, you must make it the same size, or slightly smaller, with the SCREEN command. Use 1,1 as the last two parameters to SCREEN, to keep its top left corner fixed. gORDER 1,1 will then bring the default window to the front, and with it the text window. EDIT can then be used.

Here is an example program which uses this technique- moving an 'edit box', hiding it while you edit, then finally letting you move it around.

```
PROC gsetwl:
   LOCAL a$(100),w%,h%,g$(1),factor%
   w%=gWIDTH :h%=gHEIGHT
   gSETWIN w%/4,h%/4,138,9 REM default window...
   SCREEN 23,1,1,1 REM and text window
   PRINT "Text win:"; :GET
   gCREATE(0,0,w%,h%,1) REM new large window
   gPATT -1,gWIDTH,gHEIGHT,0 REM shade it
   gAT 2,15 :gTMODE 4
   gPRINT "Graphics window 2"
   gORDER 1,0 REM back to default+text window
   PRINT REM PRINT uses text window
   PRINT "Textwin:";
   EDIT a$ REM the edit itself
   gORDER 1,9 REM to background
   CLS
   a$=""
   PRINT "Hidden:"; :EDIT a$ REM can't see this
   gORDER 1,0 :GET REM now here it is
   gUSE 1 REM graphics go to default window
   DO  REM move default/text window around
      CLS
      PRINT "Up,Down,Left,Right,Quit";
      g$=UPPER$(GET$)
      IF kmod=2 REM Shift key moves quickly
         factor%=10
      ELSE
         factor%=1
      ENDIF
      IF g$="U"
         gSETWIN gORIGINX,gORIGINY-factor%
      ELSEIF g$="D"
         gSETWIN gORIGINX,gORIGINY+factor%
      ELSEIF g$="L"
         gSETWIN gORIGINX-factor%,gORIGINY
      ELSEIF g$="R"
         gSETWIN gORIGINX+factor%,gORIGINY
      ENDIF
   UNTIL g$="Q" REM Q to quit
ENDP
```



7

Friendlier interaction

Everyday OPL programs can use the same graphical interface seen throughout the Series 3:

- ***Menus* offer lists of options for you to choose from. You can also select these options with *hot-keys* like PSION-A, PSION-B etc.**

- ***Dialogs* let a program ask for all kinds of information - numbers, filenames, dates and times etc. - in one go.**

- **The *status window*, and screen messages such as 'Busy' are also available.**

Menu keywords begin with an `M`**, and dialog keywords with a** `D`**. In this manual a lower case is used for these letters - for example,** `mINIT` **and** `dEDIT` **- to distinguish them from other commands such as** `MINUTE` **and** `DELETE`**. As with other OPL commands, though, you can type them using upper or lower case letters, so** `MINIT`**,** `minit` **and** `mINIT` **all mean the same thing.**

Menus

Menus provide a simple way for any reasonably complex OPL program to let you choose from its various options.

To display menus in OPL takes three steps:

- Use the mINIT command. This prepares OPL for new menus.
- Use the mCARD command to define each menu.
- Use the MENU function to display the menus.

You use the displayed menus like any others on the Series 3. Use the arrow keys to move around the menus. Press Enter (or an option's hot-key) to select an option, or press Esc to cancel the menus without making a choice. In either case, the menus are removed, the screen redrawn as it was, and MENU returns a value to indicate the selection made.

Defining the menus

The first parameter to mCARD is the name of the menu. This will appear at the top of the menu; the names of all of the menus form a bar across the top of the screen.

From one to six items on the menu may be defined, each specified by two parameters. The first is the item name, and the second the keycode for a hot-key. This specifies a key which, when pressed together with the PSION key, should select the option. (Your program must still handle hot-keys which are pressed without using the menu.) It is easiest to specify the hot-key with `%` - for example, `%A` gives the value for A. (You can use the upper or lower case value for the key.)

For example, mCARD `"Comms","Setup",%S,"Transfer",%T` defines a menu with the title `Comms`. When you use move to this menu using ← or →, you'll see it has the two options `Setup` and `Transfer`, with hot-keys PSION-S and PSION-T respectively.

Each subsequent mCARD defines the next menu to the right. A large OPL application might use mCARD like this:

```
mCARD "File","New",%N,"Open",%O,"Save",%S
mCARD "Edit","Copy",%C,"Insert",%I,"Eval",%E
mCARD "Search","First",%F,"Next",%G,"Previous",%D
```

Displaying the menus

The MENU function displays the menus defined by mINIT and mCARD, and waits for you to select an item. It returns the hot-key keycode of the item selected, in lower case, whether you used Enter or the hot-key itself to select it.

Important: any hot-key keypress will be returned - even those not in the menu.

If you cancel the menus by pressing Esc, MENU returns 0.

Problems with menus

When choosing hot-keys, do not use those such as the number keys which produce different characters when used with the PSION key. Unless you have a good reason not to, stick with A to Z.

You must ensure that you do not use the same hot-key twice when defining the menus, as OPL does not check for this.

Each menu definition uses some memory, so 'Memory full' errors are possible.

Don't forget to use mINIT before you begin defining the menus.

If the menus defined by mCARD are too wide in total to fit on the screen, MENU will raise an error.

The 'last used' option is not remembered when you use a menu a second time.

A menu example

This procedure allows you to press the Menu key and see a menu. You might instead be typing a number or some text into the program, or moving around in some way with the arrow keys, and this procedure returns any such keypresses. You could use this procedure instead of a simple GET whenever you want to allow a menu to be shown, and its hot-keys to work.

Each option in the menus has a corresponding procedure named `proc` plus the hot-key letter - so for example, the option with hot-key PSION-N is handled by the procedure `procn`.

This procedure uses the technique of calling procedures by strings, as described in the chapter on calling procedures.

```
PROC kget%:
   LOCAL k%,h$(9),a$(5)
   h$="NOSCIEFGD" REM our hot-keys
   WHILE 1
      k%=GET
      IF k%=$122 REM Menu key
         mINIT
         mCARD "File","New",%N,"Open",%O,"Save",%S
         mCARD "Edit","Copy",%C,"Insert",%I,"Eval",%E
         mCARD "Search","First",%F,"Next",%G,"Previous",%D
         k%=MENU
         IF k% AND INTF(LOC(h$,CHR$(k%)))
            a$="proc"+CHR$(k%)
            @(a$): REM procn:, proco:, ...
         ENDIF
      ELSEIF k% AND $200 REM hot-key (PSION key used)
         k%=(k%-$200) AND $FFDF REM Uppercase
         k%=LOC(h$,CHR$(k%)) REM One of ours?
         IF k%
            a$="proc"+MID$(h$,k%,1)
            @(a$): REM procn:, proco:, ...
         ENDIF REM ignore other weird keypresses
      ELSE REM some other key
         RETURN k%
      ENDIF
   ENDWH
ENDP

PROC procn:
...
ENDP

PROC proco:
...
ENDP

...
```

Dialogs

In OPL, dialogs are constructed in a similar way to menus:

- Use the dINIT command to prepare OPL for a new dialog. If you give a string parameter to dINIT it will be displayed as a title for the dialog, separated from the rest of the dialog by a horizontal line.
- Define each line of the dialog, from top to bottom. There are separate commands for each type of item you can use in a dialog - for example, dEDIT for editing a string, dDATE for typing in a date, and so on.
- Use the DIALOG function to display the dialog. It returns a number indicating the line you were on when you pressed Enter (counting any title line as line 1), or 0 if you pressed Esc.

Use ↑ and ↓ to move from line to line, and enter the relevant information, as in any other Series 3 dialog. You can even press Tab to produce vertical lists of options.

Each of the commands like dEDIT and dDATE specifies a variable to take the information you type in. If you press Enter to complete the dialog, the information is saved in those variables. The dialog is then removed, and the screen redrawn as it was.

You can press Esc to abandon the dialog without making any changes to the variables.

If you enter information which is not valid for the particular line of the dialog, you will be asked to re-enter different information.

Here is simple example. It assumes a global variable `name$` exists:

```
PROC getname:
   dINIT "Who are you?"
   dEDIT name$,"Name:"
   DIALOG
ENDP
```

This procedure displays a dialog with `Who are you?` as its top-line title, and an edit box for typing in your name. If you end by pressing Enter, the name you have typed will be saved in `name$`; if you press Esc, `name$` is not changed.

When the dialog is first displayed, the existing contents of `name$` are used as the string to edit.

Note that the dialog is automatically created with a width suitable for the item(s) you defined, and is centred in the screen.

Lines you can use in dialogs

This section describes the various commands that can define a line of a dialog. In all cases:

- `prompt$` is the string which will appear on the left side of the line.
- *`var`* denotes an argument which **must** be a LOCAL or GLOBAL variable, **because it takes the value you enter.** Single elements of arrays may also be used, but not field variables or procedure parameters. ('`var`' is just to show you where you must use a suitable variable - you don't actually type it.)

  Where appropriate, this variable provides the initial value shown in the dialog.

Although examples are given using each group of commands, you can mix commands of any type to make up your dialog.

More details of the commands may be found in the alphabetic listing.

Strings, secret strings and filenames

`dEDIT` *`var str$,prompt$,len%`*
defines a string edit box.

`len%` is an optional parameter. If supplied, it gives the width of the edit box (allowing for widest possible character in the font). The string will scroll inside the edit box, if necessary. If `len%` is not supplied, the edit box is made wide enough for the maximum width `str$` could be.

`dXINPUT` *`var str$,prompt$`*
defines a secret string edit box, such as for a password. A special symbol will be displayed for each character you type, to preserve the secrecy of the string.

`dFILE` *`var str$,prompt$,f%`*
defines a filename edit box.

Here is an example dialog using these three commands:

```
PROC info:
   dINIT "Your personal info"
   dEDIT n$,"Name:",15
   dXINPUT pw$,"Password:"
   dFILE f$,"Log file:",0
   RETURN DIALOG
ENDP
```

This returns 'True' if Enter was used, indicating that the GLOBAL variables `n$`, `pw$` and `f$` have been updated.

dFILE automatically has a 'Disk' selector on the line below it. The third parameter to dFILE controls the type of file editor you see, and the kind of input allowed. See the 'Alphabetic listing' for more details of dFILE.

Choosing one of a list

`dCHOICE var choice%,prompt$,list$`
defines a choice list. `list$` should contain the possible choices, separated by commas - for example, `"Yes,No"`. The `choice%` variable specifies which choice should initially be shown - 1 for the first choice, 2 for the second, and so on.

For example, here is a simple "choice" dialog:

```
PROC dcheck:
   LOCAL c%
   c%=1       REM default to "Internal"
   dINIT "Disk Check"
   dCHOICE c%,"Disk:","Internal,A,B"
   IF DIALOG     REM returns 0 if cancelled
      ... REM disk-check code
   ENDIF
ENDP
```

Numbers, dates and times

`dLONG var long&,prompt$,min&,max&`
and
`dFLOAT var fp,prompt$,min,max`
define edit boxes for long integers and floating-point numbers respectively. Use dFLOAT to allow fractions, and dLONG to disallow them. `min(&)` and `max(&)` give the minimum and maximum values which are to be allowed. There is no separate command for ordinary integers - use dLONG with suitable `min&` and `max&` values.

`dDATE var long&,prompt$,min&,max&`
and
`dTIME var long&,prompt$,type%,min&,max&`
define edit boxes for dates and times. `min&` and `max&` give the minimum and maximum values which are to be allowed.

For dDATE, `long&`, `min&` and `max&` are specified in "days since 1/1/1900". The DAYS function is useful for converting to "days since 1/1/1900".

For dTIME, long&, min& and max& are in "seconds since 00:00". The DATETOSECS and SECSTODATE functions are useful for converting to and from "seconds since midnight" (they actually use "seconds since 00:00 on 1/1/1970").

dTIME also has a type% parameter. This specifies the type of display required:

type%	*time display*
0	absolute time without seconds
1	absolute time with seconds
2	duration without seconds
3	duration with seconds

For example, 03:45 represents an absolute time while 3 hours 45 minutes represents a duration.

This procedure creates a dialog, using these commands:

```
PROC delivery:
   LOCAL d&,t&,num&,wt
   d&=DAYS(DAY,MONTH,YEAR)
   DO
      t&=secs&:
   UNTIL t&=secs&:
   num&=1 :wt=10
   dINIT "Delivery"
   dLONG num&,"Boxes",1,1000
   dFLOAT wt,"Weight (kg)",0,10000
   dDATE d&,"Date",d&,DAYS(31,12,1999)
   dTIME t&,"Time",0,0,DATETOSECS(1970,1,1,23,59,59)
   IF DIALOG    REM returns 0 if cancelled
      ... REM rest of code
   ENDIF
ENDP

PROC secs&:
   RETURN HOUR*INT(3600)+MINUTE*60
ENDP
```

The secs&: procedure uses the HOUR and MINUTE functions, which return the time as kept by the Series 3. It is called twice to guard against an incorrect result, in the (albeit rare) case where the time ticks past the hour between calling HOUR and calling MINUTE.

The INT function is used in secs&: to force OPL to use long integer arithmetic, avoiding the danger of an 'Integer overflow' error.

d& and t& are set up to give the current date and time when the dialog is first displayed. The value in d& is also used as the minimum value for

dDATE, so that in this example you cannot set a date before the current date.

DATETOSECS is used to give the number of seconds representing the time 23:59. The first three parameters, `1970`, `1` and `1`, represent the first day from which DATETOSECS begins calculating.

★ *Results from dDATE*

dDATE returns a value as a number of days. To convert this to a date:

- If you are dealing only with days on or after 1/1/1970, you can subtract 25567 (`DAYS(1,1,1970)`), multiply by 86400 (the number of seconds in a day), and use SECSTODATE.
- To handle days before 1/1/1970 as well, you can call the Operating System to perform the conversion. This procedure is passed one parameter, the number of days, and from it sets four global variables - `day%`, `month%`, `year%` and `yrdy%`. It calls the Operating System with the OS function:

```
PROC daytodat:(days&)
   LOCAL dyscent&(2),dateent%(4)
   LOCAL flags%,ax%,bx%,cx%,dx%,si%,di%
   dyscent&(1)=days&
   si%=ADDR(dyscent&()) :di%=ADDR(dateent%())
   ax%=$0600 REM TimDaySecondsToDate fn.
   flags%=OS($89,ADDR(ax%)) REM TimManager int.
   IF flags% AND 1
      RAISE (ax% OR $ff00)
   ELSE
      year%=PEEKB(di%)+1900
      month%=PEEKB(di%+1)+1
      day%=PEEKB(di%+2)+1
      yrdy%=PEEKW(di%+6)+1
   ENDIF
ENDP
```

If you do use this procedure, be careful to type it exactly as shown here.

▮ *Displaying text*

`dTEXT prompt$,body$,type%`
defines `prompt$` to be displayed on the left side of the line, and `body$` on the right. **There is no variable associated with dTEXT.** If you use a null string (`""`) for `prompt$`, `body$` is displayed across the whole width of the dialog.

`type%` is an optional parameter. If specified, it controls the alignment of `body$`:

type%	effect
0	left align body$
1	right align body$
2	centre body$

In addition, you can add any or all of the following three values to `type%`, for these effects:

type%	effect
$100	use bold text for body$.
$200	draw a line below this item.
$400	make this line selectable. (It will also be bulleted if prompt$ is not "".)

dTEXT is not just for displaying information. Since DIALOG returns a number indicating the line you were on when you pressed Enter (or 0 if you pressed Esc), you can use dTEXT to offer a choice of options, rather like a menu:

```
PROC selact:
    dINIT "Select action"
    dTEXT "","Add",$402
    dTEXT "","Copy",$402
    dTEXT "","Review",$402
    dTEXT "","Delete",$402
    RETURN DIALOG
ENDP
```

In each case `type%` is $402 ($400+2). The `$400` makes each text string selectable, allowing you to move the cursor onto it, while 2 makes each string centred.

Displaying exit keys

Most dialogs are completed by pressing Enter to confirm the information typed, or Esc to cancel the dialog. These keys are not usually displayed as part of the dialog.

However, some Series 3 dialogs offer you a simple choice, by showing pictures of the keys you can press. A simple "Are you sure?" dialog might, for example, show the two keys 'Y' and 'N', and indicate the one you press.

If you want to display a message and offer Enter, Esc and/or Space as the exit keys, you can display the entire dialog with the ALERT function.

If you want to use other keys, such as Y and N, or display the keys below other dialog items such as dEDIT, create the dialog as normal and use the dBUTTONS command to define the keys.

ALERT and dBUTTONS are explained in detail in the alphabetic listing.

Other dialog information

Positioning dialogs

If a dialog overwrites important information on the screen, you can position it with the dPOSITION command. Use dPOSITION at any time between dINIT and DIALOG.

dPOSITION uses two integer values. The first specifies the horizontal position, and the second, the vertical. `dPOSITION -1,-1` positions to the top left of the screen; `dPOSITION 1,1` to the bottom right; `dPOSITION 0,0` to the centre, the usual position for dialogs.

`dPOSITION 1,0`, for example, positions to the right hand edge of the screen, and centres the dialog half way up the screen.

Restrictions on dialogs

The following general restrictions apply to all dialogs:

- Only one dialog may be in use at a time.
- A dialog must be initialised (dINIT), defined (dEDIT etc) and displayed (DIALOG) in the same procedure.

- A dialog may consist of up to seven lines, including any title. Filename editors count as two lines, and exit keys count as three. A 'Too many items' error is raised if this limit is exceeded.
- If the width of any line would make the dialog too wide, a 'Too wide' error is raised when DIALOG is called.

The status window and screen messages

Status window - temporary or permanent?

Pressing PSION-Menu when an OPL program is running will always display a temporary status window. This status window is in front of all the OPL windows, so your program can't write over it.

Use STATUSWIN ON to display a permanent status window. It will be displayed until you use STATUSWIN OFF.

Important: The permanent status window is **behind** all other OPL windows. In order to see it, you **must** use SCREEN and gSETWIN to reduce the size of the text and graphics windows. You should ensure that your program does not create windows over the top of it.

You might use STATUSWIN ON when Control-Menu is pressed, for consistency with the rest of the Series 3.

What the status window does

The status window displays an OPL icon, the OPL program name, and a digital or analog clock according to the system setting. It is displayed on the right-hand side of the screen.

The status window is inaccessible to, and does not affect, OPL keywords such as gORDER and gRANK.

You can set or change the name displayed in the status window with SETNAME - for example, SETNAME "Bus Map" or SETNAME a$.

Information messages

GIPRINT displays an information message for 2 seconds, in the bottom right corner of the screen. For example, `GIPRINT "Not Found"` displays `Not Found`. The string you specify can be up to 63 characters. If a string is too long for the screen, it will be clipped.

You can add an integer parameter to control the corner in which the message appears:

value	corner
0	top left
1	bottom left
2	top right
3	bottom right

For example, `GIPRINT "Who?",0` prints `Who?` in the top left corner.

Only one message can be shown at a time. You can make the message go away - for example, if a key has been pressed - with `GIPRINT ""`.

'Busy' messages

Messages which say a program is temporarily busy, or cannot respond for some reason, are by convention shown in the bottom left corner. The BUSY command lets you display your own messages of this sort. Use `BUSY OFF` to remove it.

`BUSY "Paused..."`, for example, displays `"Paused..."` in the bottom left corner. This remains shown until `BUSY OFF` is used.

You can control the corner used in the same way as for GIPRINT. You can also add a third parameter, to specify a delay time (in half seconds) before the message should be shown. Use this to prevent 'busy' messages from continually appearing very briefly on the screen.

For example, `BUSY "Wait:",1,4` will display `Wait:` in the bottom left corner after a delay of 2 seconds. As soon as your program becomes responsive to the keyboard, it should use `BUSY OFF`. If this occurs within two seconds of the original BUSY, no message is seen.

Only one message can be shown at a time. The string to display can be up to 19 characters long.

8

OPL and Solid State Disks

Types of Solid State Disk

Solid State Disks (*SSDs*) are explained in detail in the User guide. There are two main reasons for using them:

- To provide more room for storing information.
- To make backup copies of important data, in case you accidentally change or delete it, or even lose your Series 3.

There are two types - *Ram SSDs* and *Flash SSDs*. They fit into either of the *SSD drives* marked `A` and `B`, at either side of the Series 3.

- Flash SSDs are for storing or backing up data which is no longer being changed. This includes finished OPL programs.
- Ram SSDs are for storing or backing up changing data. This includes OPL programs you are still writing or testing.

You can, though, save programs and data files to either kind of SSD, as you see fit.

How to put programs on an SSD

To create a new OPL module on an SSD, use the 'New file' option in the System screen as before, but set the `Disk` line of the dialog to `A` or `B` as required.

To copy an OPL module onto an SSD, move onto the module name where it is listed under the Program icon, and use the 'Copy file' option on the 'File' menu. Set the 'To Disk' line to `A` or `B`. If you want this copy to have a different name to the original, type the name to use, on the 'To file Name' line. The new copy will appear in the list under the Program icon, but with `[A]` or `[B]` after its name.

To copy the **translated** version of an OPL module, move onto the name in the list under the RunOpl icon, then proceed as before.

SSDs from inside OPL

Your OPL programs can create or use data files on SSDs. To do so, begin the name of the data file with `A:` or `B:` - for example:

```
CREATE "B:JKQ",A,X1$,X2$
```

tries to create a data file "JKQ" on an SSD in `B`, while

```
DELETE "A:X300"
```

tries to delete a data file "X300" on an SSD in `A`.

Don't confuse the drive names A and B with the logical names A, B, C and D. Logical names are unaffected by which drive a data file is on.

The internal memory can be referred to as M:, if required. For example:

```
PROC delx300:
   LOCAL a$(3),c%
   a$="MAB" :c%=1 REM default to "Internal"
   dINIT "Delete X300 data file"
   dCHOICE c%,"Disk:","Internal,A,B"
   IF DIALOG REM returns 0 if cancelled
      DELETE MID$(A$,c%,1)+":X300"
   ENDIF
ENDP
```

In this example, `MID$(A$,c%,1)` gives "M", "A" or "B", according to the choice made in the dialog. This is added on the front of ":X300" to give the name of the file to delete - "M:X300", "A:X300" or "B:X300".

When using data files with SSDs, follow the same guidelines as with OPL programs - Flash SSDs are for one-off or "finished" data, while Ram SSDs are for data which is still being changed.

Directories and DOS structure

The internal memory and SSDs use a DOS-compatible directory structure, the same as that used on disks on business PCs. For more details, see the 'Advanced Topics' chapter.

9

Example programs

This chapter contains example programs written in OPL. The programs are not intended to demonstrate all the features of OPL, but they should give you a few hints. To find out more about a particular command or function, refer to the alphabetic listing.

There are some other example programs in the 'Advanced topics' chapter.

When you're typing in

- You can type procedures in all uppercase, all lowercase or any mixture of the two. Be careful with character codes, though - `%A` is different to `%a`.
- When there is more than one command or function on a line, separate each one with a space and colon - for example:

```
CLS :PRINT "hello" :GET
```

  However, the colon is optional before a REM statement - for example:

```
CLS REM Clears the screen
```

  and

```
CLS :REM Clears the screen
```

  are both OK.
- Put a space between a command and the arguments which follow it - for example `PRINT a$`. But don't put a space between a function and the arguments in brackets - for example `CHR$(16)`.
- It doesn't matter how many spaces or tabs you have at the beginning of a line.

Errors

The following programs do not include full error handling code. This means that they are shorter and easier to understand, but may fail if, for example, you enter the wrong type of input to a variable.

If you want to develop other programs from these example programs, it is recommended that you add some error handling code to them. See the chapter on error handling for further details.

Countdown Timer

```
PROC timer:
   LOCAL min&,sec&,secs&,i%
   dINIT "Countdown timer"
   dLONG min&,"Minutes",0,&7FFFFFFF/60-1
   dLONG sec&,"Seconds",1,59
   dBUTTONS "Cancel",-27,"Start",13
   IF DIALOG=13
      STATUSWIN ON
      SCREEN 30,9,1,1
      gSETWIN 0,0,179,79
      secs&=sec&+60*min&
      WHILE secs&
         PAUSE -20
         REM a key gets us out
         IF KEY
            RETURN
         ENDIF
         secs&=secs&-1
         AT 13,4 :PRINT GEN$(secs&/60,-2);"m"
         AT 17,4 :PRINT GEN$(mod&:(secs&,int(60)),-2);"s"
      ENDWH
      DO
         BEEP 5,300
         PAUSE 10
         IF KEY :BREAK :ENDIF
         i%=i%+1
      UNTIL i%=10
   ENDIF
ENDP

PROC mod&:(a&,b&)
   REM modulo function
   REM computes (a&)mod(b&)
   RETURN a&-(a&/b&)*b&
ENDP
```

Dice

When the program is run, a message is displayed saying that the dice is rolling. You then press S to stop it. A random number from one to six is displayed and you choose whether to roll again or not.

```
PROC dice:
   LOCAL dice%
   DO      CLS :PRINT "DICE ROLLING:"
      AT 1,3 :PRINT "Press a key to stop"
      DO
         dice%=(RND*6+1)
         AT 1,2 :PRINT dice%
      UNTIL KEY
      BEEP 5,300
      dINIT "Roll again?"
      dBUTTONS "No",%N,"Yes",%Y
   UNTIL DIALOG<>%y
ENDP
```

Random numbers

This is how `dice:` displays 1, 2, 3, 4, 5 or 6 randomly. The RND function returns a random floating-point number, between 0 and 1 (not including 1). It is then multiplied by 6 and 1 is added (so that you get numbers from 1 to 6 instead of from 0 to 5). It is rounded down to a whole number by assigning to the integer `dice%`.

Birthdays

This procedure finds out on which day of the week people were born.

```
PROC Birthday:
   LOCAL day&,month&,year&,DayInWk%
   DO
      dINIT
      dTEXT "","Enter your date of birth",2
      dTEXT "","Use numbers, eg 23 12 1963",$202
      dLONG day&,"Day",1,31
      dLONG month&,"Month",1,12
      dLONG year&,"Year",1900,2155
      IF DIALOG=0
         BREAK
      ENDIF
      DayInWk%=DOW(day&,month&,year&)
      CLS :PRINT DAYNAME$(DayInWk%),
      PRINT day&,month&,year&
      dINIT "Again?"
      dBUTTONS "No",%N,"Yes",%Y
   UNTIL DIALOG<>%y
ENDP
```

The DOW function works out what day of the week, from 1 to 7, a date is. The DAYNAME$ function then converts this number to MON, TUE and so on. MON is 1 and SUN is 7.

Bouncing Ball

```
PROC bounce:
   LOCAL posX%,posY%,changeX%,changeY%,k%
   LOCAL scrx%,scry%
   scrx%=40 :scry%=9
   posX%=1 :posY%=1
   changeX%=1 :changeY%=1
   DO
      posX%=posX%+changeX%
      posY%=posY%+changeY%
      IF posX%=1 OR posX%=scrx%
         changeX%=-changeX%
         REM at edge ball changes direction
         BEEP 2, 600 REM low beep
      ENDIF
      IF posY%=1 or posY%=scry% REM same for y
         changeY%=-changeY%
         BEEP 2, 200 REM high beep
      ENDIF
      AT posX%,posY% :PRINT "O";
      PAUSE 2  REM Try changing this
      AT posX%,posY% :PRINT " ";
      REM removes old 'O' character
      k%=KEY
   UNTIL k%
ENDP
```

Data files

The following module works on a data file called DATA, containing names, addresses, post codes and telephone numbers. It assumes this file has already been created with a statement like this:

```
CREATE "DATA",A,nm$,ad1$,ad2$,ad3$,ad4$,tel$
```

To use the DATA file which the Data button uses, you need to open "\DAT\DATA.DBF".

The first procedure is the controlling, calling procedure, offering you choices. The next two let you add or edit records.

```
PROC files:
   GLOBAL nm$(255),ad1$(255),ad2$(255)
   GLOBAL ad3$(255),ad4$(255),tel$(255),title$(30)
   LOCAL g%
   OPEN "DATA",A,nm$,ad1$,ad2$,ad3$,ad4$,tel$
   DO
      CLS
      dINIT "Select action"
      dTEXT "","Add new record",$402
      dTEXT "","Find and edit a record",$402
      g%=DIALOG
      IF g%=2
         add:
      ELSEIF g%=3
         edit:
      ENDIF
   UNTIL g%=0
   CLOSE
ENDP


PROC add:
   nm$="" :ad1$="" :ad2$=""
   ad3$="" :ad4$="" :tel$=""
   title$="Enter a new record"
   IF showd%:
      APPEND
   ENDIF
ENDP
```

```
PROC edit:
   LOCAL search$(30),p%
   dINIT "Find and edit a record"
   dEDIT search$,"Search string",15
   IF DIALOG
      FIRST
      IF FIND("*"+search$+"*")=0
         ALERT("No matching records")
         RETURN
      ENDIF
      DO
         nm$=A.nm$ :ad1$=A.ad1$ :ad2$=A.ad2$
         ad3$=A.ad3$ :ad4$=A.ad4$ :tel$=A.tel$
         title$="Edit matching record"
         IF showd%:
            UPDATE :BREAK
         ELSE
            NEXT
         ENDIF
         FIND("*"+search$+"*")
         IF EOF
            ALERT("No more matching records")
            BREAK
         ENDIF
      UNTIL 0
   ENDIF
ENDP

PROC showd%:
   LOCAL ret%
   dINIT title$
   dEDIT nm$,"Name",25
   dEDIT ad1$,"Street",25
   dEDIT ad2$,"Town",25
   dEDIT ad3$,"County",25
   dEDIT ad4$,"Postcode",25
   dEDIT tel$,"Phone",25
   ret%=DIALOG
   IF ret%
      A.nm$=nm$ :A.ad1$=ad1$ :A.ad2$=ad2$
      A.ad3$=ad3$ :A.ad4$=ad4$ :A.tel$=tel$
   ENDIF
   RETURN ret%
ENDP
```

Re-order

When you enter or change a record, it goes to the end of the address book. However, if each entry in your address book begins with a person's second name - for example, `Tate, Sharon` - you can use this program to re-order the address book. This doesn't change the way you find a record, but after running it you can step through it like a paper address book, or print it out neatly ordered.

This procedure can be used as required for any data file in internal memory or on Ram SSDs. If used on a data file held on a Flash SSD it would use up disk space each time you run it. The dialog it shows is set to show data files used by the Database.

You can adapt this procedure to sort other types of data files in other ways.

```
PROC reorder:
   LOCAL last%,e$(255),e%,lpos%,n$(128),c%
   n$="\dat\*.dbf"
   dINIT "Re-order Data file"
   dFILE n$,"Filename",0
   IF DIALOG REM returns 0 if cancelled
      OPEN n$,a,a$
      LAST :last%=POS
      IF COUNT>0
         WHILE last%<>0
            POSITION last% :e%=POS
            e$=UPPER$(a.a$)
            DO
               IF UPPER$(a.a$)<e$
                  e$=UPPER$(a.a$) :e%=POS
               ENDIF
               lpos%=POS :BACK
            UNTIL pos=1 and lpos%=1
            POSITION e%
            PRINT e$
            UPDATE :last%=last%-1
         ENDWH
      ENDIF
      CLOSE
   ENDIF
ENDP
```

Stopwatch

Here is a simple stopwatch with lap times. Note that the Series 3 switches off automatically after a time if no keys are pressed; you may want to disable this feature before running this program.

Each timing is only accurate to within one second, as the procedure is based on the SECOND function.

```
PROC watch:
   LOCAL k%,s%,se%,mi%
   AT 15,1 :PRINT "Stopwatch"
   AT 10,9 :PRINT "Press a key to start"
   GET
   DO
   CLS :mi%=0:se%=0:s%=SECOND
   AT 10,9 :PRINT "   S=Stop, L=Lap    "
   loop::
   k%=KEY AND $ffdf REM ensures upper case
   IF k%=%S
      GOTO pause::
   ENDIF
   IF k%=%L
      AT 15,5 :PRINT "Lap: ";mi%;":";
      IF se%<10 :PRINT "0"; :ENDIF
      PRINT se%;" ";
   ENDIF
   IF SECOND<>s%
      s%=SECOND :se%=se%+1
      IF se%=60 :se%=0:mi%=mi%+1 :ENDIF
      AT 13,7
      PRINT "Mins",mi%,"Secs",
      IF se%<10 :PRINT "0"; :ENDIF
      PRINT se%;" ";
   ENDIF
   GOTO loop::
   pause::
   mINIT
   mCARD "Watch","Restart",%R,"Zero",%Z,"Exit",%X
   k%=MENU
   IF k%=%r
      GOTO loop::
   ENDIF
   UNTIL k%<>%z
ENDP
```

Animation example

This program requires five bitmap files - `one.pic` to `five.pic`. Each of these would differ slightly - they might, for example, be five 'snapshots' of a running human figure, each with the legs at a different point in their cycle.

The program copies each bitmap into a window of its own, then makes each window visible in turn, each time slightly further across the screen.

To make bitmap files, first draw the pattern you want with any of the graphics drawing commands. (Use `gLINEBY 0,0` to draw single dots.) When the pattern is complete, use `gSAVEBIT` to make the bitmap file.

```
PROC animate:
LOCAL id%(5),i%,j%,s$(5,10),w%,h%
w%=16 :h%=28 REM example width and height
s$(1)="one" :s$(2)="two" :s$(3)="three"
s$(4)="four" :s$(5)="five"
j%=1
WHILE j%<6
   i%=gLOADBIT(s$(j%))
   id%(j%)=gCREATE(0,0,w%,h%,0)
   gCOPY i%,0,0,w%,h%,3
   gCLOSE i%
   j%=j%+1
ENDWH
i%=0 :gorder 1,9
DO
   j%=(i%-5*(i%/5))+1 REM (i% MOD 5)+1
   gVISIBLE OFF REM previous window
   gUSE id%(j%) REM new window
   gSETWIN i%,20 REM position it
   gORDER id%(j%),1 REM make foreground
   gVISIBLE ON REM make visible
   i%=i%+1
   PAUSE 2
UNTIL KEY OR (i%>(240-w%)) REM screen edge
ENDP
```

Inserting a new line in a database

If you insert a new label in a database, the entries will no longer match up with the labels. Rather than using the 'Update' option on every entry, to insert a suitable blank line in each one, you can use this program to do this for the entire data file.

The Database allows you to use as many lines (fields) as you want in an entry (record); OPL can only access 32 fields. This program only lets you insert a new field in the first 16 fields, although you can adapt the program simply to check up to 31 fields.

If, in the Database, you enter a line longer than 255 characters, it is stored as two fields, with a character of code 20 at the start of the second field. This program correctly handles any such fields.

The program checks that the 17th field is blank, as it will be overwritten by what was the 16th field. If a long entry has a 17th field, and it contains text, the program skips this entry. The rest of longer entries - even if there are more than 32 fields - will be unchanged.

If you insert a new field at a position below the last label, the Database will not show it, even when using 'Update'.

The maximum record length in OPL is 1022 characters. The OPEN command will display a 'Record too large' error if the file contains a record longer than this.

```
PROC label:
LOCAL a%,b%,c%,d%,s$(128),s&,i$(17,255)
s$="\dat\*.dbf"
dINIT "Insert new field"
dFILE s$,"Data file",0
dLONG s&,"Break at line (1-16)",1,16
IF DIALOG
OPEN s$,A,a$,b$,c$,d$,e$,f$,g$,h$,i$,j$,k$,l$,m$,n$,o$,p$,q$
   c%=COUNT :a%=1
   WHILE a%<=c%
      AT 1,1 :PRINT "Entry",a%,"of",c%,
      IF A.q$="" REM Entry (hopefully) not too long
         i$(1)=A.a$ :i$(2)=A.b$ :i$(3)=A.c$ :i$(4)=A.d$
         i$(5)=A.e$ :i$(6)=A.f$ :i$(7)=A.g$ :i$(8)=A.h$
         i$(9)=A.i$ :i$(10)=A.j$ :i$(11)=A.k$ :i$(12)=A.l$
         i$(13)=A.m$ :i$(14)=A.n$ :i$(15)=A.o$ :i$(16)=A.p$
         d%=0 :b%=0
         WHILE d%<s&+b% REM find field to break at
            d%=d%+1
            IF LEFT$(i$(d%),1)=CHR$(20) REM line>255...
               b%=b%+1 REM ...so it's 2 fields
            ENDIF
         ENDWH
         b%=17
         WHILE b%>d% REM copy the fields down
            i$(b%)=i$(b%-1) :b%=b%-1
         ENDWH
         i$(d%)="" REM and make an empty field
         A.a$=i$(1) :A.b$=i$(2) :A.c$=i$(3) :A.d$=i$(4)
         A.e$=i$(5) :A.f$=i$(6) :A.g$=i$(7) :A.h$=i$(8)
         A.i$=i$(9) :A.j$=i$(10) :A.k$=i$(11) :A.l$=i$(12)
         A.m$=i$(13) :A.n$=i$(14) :A.o$=i$(15) :A.p$=i$(16)
         A.q$=i$(17)
      ELSE
         PRINT "has too many fields"
         PRINT "Press a key..." :GET
      ENDIF
      UPDATE :FIRST
      a%=a%+1
   ENDWH :CLOSE
ENDIF
ENDP
```

Circles

Here are two example programs for drawing circles - the first hollow, the second filled:

```
PROC circle:
LOCAL a%(963),c&,d%,x&,y&,r&,h,y%,y1%,c2%
dINIT "Draw a circle"
x&=120 :dLONG x&,"Centre x pos",0,239
y&=40 :dLONG y&,"Centre y pos",0,79
r&=20 :dLONG r&,"Radius",1,120
h=1 :dFLOAT h,"Relative height",0,999
IF DIALOG
   a%(1)=x&+r& :a%(2)=y& :a%(3)=4*r&
   c&=1 :d%=2*r& :y1%=0
   WHILE c&<=d%
      c2%=c&*2 :y%=-SQR(r&*c2%-c&**2)*h
      a%(2+c2%)=-2 :a%(3+c2%)=y%-y1%
      y1%=y% :c&=c&+1
   ENDWH
   c&=1
   WHILE c&<=d%
      c2%=c&*2 :y%=SQR(r&*c2%-c&**2)*h
      a%(2+a%(3)+c2%)=2 :a%(3+a%(3)+c2%)=y%-y1%
      y1%=y% :c&=c&+1
   ENDWH
   GPOLY a%()
ENDIF
ENDP


PROC circlef:
LOCAL c&,d%,x&,y&,r&,h,y%
dINIT "Draw a filled circle"
x&=120 :dLONG x&,"Centre x pos",0,239
y&=40 :dLONG y&,"Centre y pos",0,79
r&=20 :dLONG r&,"Radius",1,120
h=1 :dFLOAT h,"Relative height",0,999
IF DIALOG
   c&=1 :d%=2*r& :gAT x&-r&,y& :gLINEBY 0,0
   WHILE c&<=d%
      y%=-SQR(r&*c&*2-c&**2)*h
      gAT x&-r&+c&,y&-y% :gLINEBY 0,2*y%
      c&=c&+1
   ENDWH
ENDIF
ENDP
```

10

Error handling

Syntax errors

Syntax errors are those which are reported when translating a procedure.

All programming languages are very particular about the way commands and functions are used, especially in the way program statements are laid out. Below are a number of errors which are easy to make in OPL. The incorrect statements are in bold and the correct versions are on the right.

Punctuation errors

Omitting the colon between statements on a multi-statement line:

Incorrect	Correct
`a$="text" PRINT a$`	`a$="text" :PRINT a$`

Omitting the space before the colon between statements :

Incorrect	Correct
`a$=b$:PRINT a$`	`a$=b$ :PRINT a$`

Omitting the colon after a called procedure name:

Incorrect	Correct
`PROC proc1:`	`PROC proc1:`
`GLOBAL a,b,c`	`GLOBAL a,b,c`
.	.
`proc2`	`proc2:`

Using only 1 colon after a label in GOTO/ONERR/VECTOR (instead of 0 or 2):

Incorrect	Correct
`GOTO below:`	`GOTO below`
.	.
`below::`	`below::`

Structure errors

The DO...UNTIL, WHILE...ENDWH and IF...ENDIF structures can produce a 'Structure fault' error if used incorrectly:

- Attempting to nest any combination of these structures more than eight levels deep.
- Mixing up the three structures - e.g. by using DO...WHILE instead of DO...UNTIL.
- Using ELSE IF with a space, instead of ELSEIF.

Errors in running procedures

OPL may display an error message and stop a running program if certain 'error' conditions occur. This may happen because:

- There is a mistake, or *bug*, in your program, which could not be detected during translation - for example, a calculation has involved a division by zero.
- A problem has occurred which prevents a command or function from working - for example, an APPEND command may fail because an SSD is full.

Unless you include statements which can handle such errors when they occur, OPL will use its own error handling mechanism. The program will stop and an error message be displayed. The first line gives the name of the procedure in which the error occurred, and which module this procedure is in. The second line is the 'error message' - one of the messages listed at the end of this chapter. If appropriate, you will also see a list of variable names or procedure names causing the error.

If you were editing the module with the Program editor and you ran it from there, you would also be taken back to editing the OPL module, with the cursor at the line where the error occurred.

Error handling functions and commands

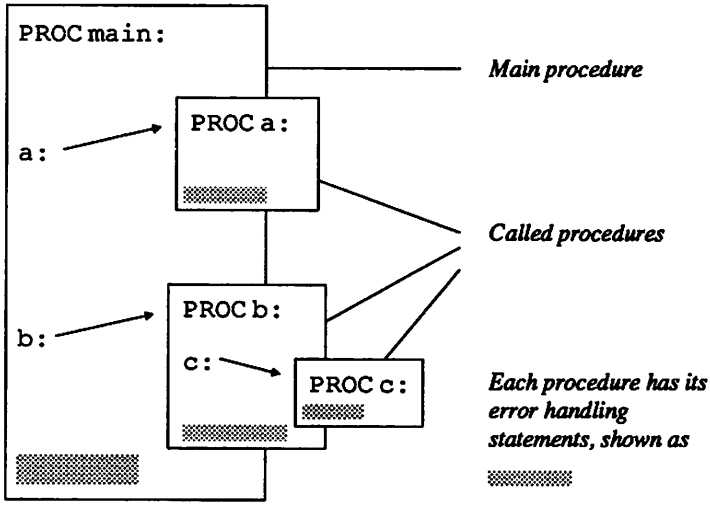
To prevent your program being stopped by OPL when an error occurs, include statements in your program which *anticipate* possible errors and take appropriate action. The following error handling facilities are available in OPL:

- TRAP temporarily suppresses OPL's error processing
- ERR and ERR$ find out what kind of error has occurred.
- ONERR establishes an error handler which can suppress OPL's error processing over whole modules.
- RAISE can be used to simulate error conditions.

These facilities put you in control and must be used carefully.

Strategy

You should design the error handling of a program in the same way as the program itself. OPL works best when programs are built up from procedures, and you should design your error handling on the same basis: each procedure should normally contain its own local error handling -



The error handling statements can then be appropriate to the procedure. For example, a procedure which performs a calculation would have one type of error handling, but another procedure which offers a set of choices would have another.

TRAP

TRAP can be used with any of these commands: APPEND, BACK, CLOSE, COMPRESS, COPY, CREATE, DELETE, ERASE, EDIT, FIRST, gCLOSE, gCOPY, gFONT, gPATT, gSAVEBIT, gUNLOADFONT, gUSE, INPUT, LAST, LCLOSE, LOADM, LOPEN, MKDIR, NEXT, OPEN, OPENR, POSITION, RENAME, RMDIR, UNLOADM, UPDATE and USE.

The TRAP command immediately precedes any of these commands, separated from it by a space - for example:

```
TRAP INPUT a%
```

If an error occurs in the execution of the command, the program does not stop, and the next line of the program executes as if there had been no error. Normally you would use ERR on the line after the TRAP to find out what the error was.

Example

When INPUT is used without TRAP and a text string is entered when a number is required, the display just scrolls up and a ? is shown, prompting for another entry. With TRAP in front of INPUT, you can handle bad entries yourself:

```
PROC trapinp:
LOCAL profit%
   DO
      PRINT
      PRINT "Enter profit",
      TRAP INPUT profit%
   UNTIL ERR=0
   PRINT "Valid number"
   GET
ENDP
```

This example uses the ERR function.

ERR and ERR$

When an error occurs in a program, check what number the error was, with the ERR function:

```
e%=ERR
```

If ERR returns zero, there was no error. The value returned by ERR is the number of the last error which occurred - it changes when a new error occurs. TRAP sets ERR to zero if no error occurred. Check the number it returns against the error messages listed at the end of this Chapter.

The ERR$ function gives you the message for error number *e%*:

```
e$=ERR$(e%)
```

You can also use ERR and ERR$ in conjunction:

```
e$=ERR$(ERR)
```

This returns the error message for the most recent error.

▮ Example

The lines below anticipate that error number -101 (File already open) may occur. If it does, an appropriate message is displayed.

```
TRAP OPEN "main",A,a$
e%=ERR
IF e% REM Checks for an error
   IF e%=-101
      PRINT "file is already open!"
   ELSE
      PRINT ERR$(e%)
   ENDIF
ENDIF
```

The inner IF...ENDIF structure displays either the message in quotes if the error was number -101, or the standard error message for any other error.

★ TRAP INPUT/EDIT and the Esc key

If in response to a TRAP INPUT or TRAP EDIT statement, the Esc key is pressed while no text is on the input/edit line, the 'Escape key pressed' error (number -114) will be raised. This error will only be raised if the INPUT or EDIT has been trapped. Otherwise, the usual ? will be presented, asking for a valid entry. It would not cause a jump to an ONERR label, for example.

You can use this feature to enable someone to press the Esc key to escape from editing or inputing a value. For example:

```
PROC trapInp:
   LOCAL a%,b%,c%
   PRINT "Enter values."
   PRINT "Press Esc to exit"
   PRINT "a% =", :TRAP INPUT a%
   IF ERR=-114 :GOTO end :ENDIF
   PRINT "b% =", :TRAP INPUT b%
   IF ERR=-114 :GOTO end :ENDIF
   PRINT "a%*b% =",a%*b%
   PAUSE -40
   RETURN
end::
   PRINT :PRINT "OK, finishing..."
   PAUSE -40
   RETURN
ENDP
```

ONERR ... ONERR OFF

ONERR sets up an error handler. This means that, whenever an error occurs in the procedure containing ONERR, the program will jump to a specified label instead of stopping in the normal way. This error handler is active until an ONERR OFF statement.

You specify the label after the word ONERR. It may be up to 8 characters long.

The label itself can then occur anywhere in the same procedure - even above the ONERR statement. After the label should come the statements handling whatever error may have caused the program to jump there. For example, you could just have the statement `PRINT ERR$(ERR)` to display the message for whatever error occurred.

All statements after the ONERR command, including those in procedures called by the procedure containing the ONERR, are protected by the ONERR, until the ONERR OFF instruction is given.

Example

```
PROC div0:
   ONERR errHand
   PRINT 1/0
   REM causes divide by zero error -8
   RETURN REM we don't get to this line
   errHand::
   ONERR OFF
   PRINT "Error:";err,err$(err)
   IF ERR = -8
   REM divide by zero error = -8
      PRINT "Division by zero is illegal"
   ENDIF
   GET
ENDP
```

Statements protected by ONERR

If an error occurs in the lines between ONERR `errHand` and ONERR OFF, the program jumps to the label `errHand::` where a message is displayed.

When to use ONERR OFF

You could protect the whole of your program with a single ONERR. However, it's often easier to manage a set of procedures which each have their own ONERR...ONERR OFF handlers, each covering their own procedure. Secondly, an endless loop may occur if all errors feed back to the same single label.

For example, the diagram below shows how two completely different errors cause a jump to the same label, and cause an inappropriate explanatory message to be displayed. An endless loop is created because `next:` is called repeatedly:

```
PROC first:
   ONERR label
   a=log(-1)
   ...
   label::
   PRINT "Log error"
   next:
ENDP

PROC next:
   PRINT 2/0
   ONERR OFF
ENDP
```

Statements protected by ONERR

Always cancel ONERR with ONERR OFF immediately after the label.

★ Multiple ONERRs

You can have more than one ONERR in a procedure, but only the most recent ONERR is active. Any errors cause a jump to the label for the most recent ONERR.

ONERR OFF disables *all* ONERRs in the current procedure. But if there are other ONERRs in procedures above this procedure (calling procedures) these ONERRs are *not* disabled.

★ TRAP and ONERR

TRAP has priority over ONERR. In other words, an error from a command used with TRAP will not cause a jump to the error handler specified with ONERR.

RAISE

The RAISE command generates an error, in the same way that OPL raises errors whenever it meets certain conditions which it recognises as unacceptable (for example, when invalid arguments are passed to a function). Once an error has been raised, either by OPL or by the RAISE command, the error-handling mechanism currently in use takes effect - the program will stop and report a message, or if you've used ONERR the program will jump to the ONERR label.

There are two reasons for using RAISE:

- You may want to mimic OPL's error conditions in your own procedures. For example, if you create a new procedure which performs a calculation and returns a value, you may want to RAISE the 'Invalid arguments' error if unsuitable numbers are passed as parameters.

  In this case, you would RAISE one of the standard error numbers. You could handle this yourself with ONERR, or let OPL handle it in the normal way.

- OPL raises only a limited range of errors for general use, and you may want to raise new kinds of error - error codes specific to your program or particular circumstances.

  In this case, you would RAISE a new error number. With ONERR on, RAISE would go to the ONERR label, where you would have code to

interpret your new error numbers. You could then display appropriate messages.

You can use any positive number as a new error code. Do not use any of the numbers in the list that follows - stay well clear of them, in case future versions of OPL extend the range of standard error codes.

You may also find RAISE useful for testing your error handling.

Example:

```
PROC main:
   REM calling procedure
   WHILE 1
      PRINT myfunc:(0.0)      REM raises error -2
   ENDWH
ENDP

PROC myfunc:(x)
   LOCAL s
   REM returns 1/sqr(x)
   s=SQR(x)
   IF s=0
      RAISE -2
      REM 'Invalid arguments'
      REM avoids 'divide by zero'
   ENDIF
   RETURN (1/s)
ENDP
```

This uses RAISE to raise the 'Invalid arguments' error not the 'divide by zero' error, since the former is the more appropriate message.

Error messages

These are the numbers of the errors which OPL can raise, and the message associated with them:

Number	*Message:*
-1	General failure
-2	Invalid arguments
-3	O/S error
-4	Service not supported
-5	Underflow (number too small)
-6	Overflow (number too large)
-8	Divide by zero
-9	In use (e.g. serial port being used by another program)
-10	No system memory
-13	Process table full (max. number of processes reached)
-14	Resource already open
-15	Resource not open
-19	Filing system not found (e.g. if you unplug cable to PC)
-20	Failed to start
-21	Font not loaded
-22	Too wide (dialogs)
-23	Too many items (dialogs)

File and device errors

-32	Already exists
-33	Does not exist
-34	Write failure
-35	Read failure
-36	End of file (when you try to read past end of file)
-37	Disk full
-38	Invalid name
-39	Access denied (e.g. to a protected file on PC)
-40	Resource locked (i.e. temporarily in use)
-41	Device does not exist
-42	Directory does not exist
-43	Record too large
-44	Read only file
-47	Invalid volume (corrupt disk)
-48	I/O cancelled
-50	Disconnected

-51	Connected
-52	Too many retries
-53	Line failure
-54	Inactivity timeout
-55	Incorrect parity
-56	Serial frame (usually because Baud setting is wrong)
-57	Serial overrun (usually because Handshaking is wrong)
-58	Cannot connect to remote modem
-59	Remote modem busy
-60	No answer (from remote modem)
-61	Black listed (you may try a number only a certain number of times; wait a while and try again)
-62	Not ready
-63	Unknown media (corrupt SSD)
-64	Root directory full (on any device, the root directory has a maximum amount of memory allocated to it)
-65	Write protected
-66	Media is corrupt
-67	User abandoned
-68	Erase pack failure
-69	Wrong file type

Translator errors

-70	Missing "
-71	String too long
-72	Unexpected name
-73	Name too long
-74	Logical device must be A-D
-75	Bad field name
-76	Bad number
-77	Syntax error
-78	Illegal character
-79	Function argument error
-80	Type mismatch
-81	Missing label
-82	Duplicate name
-83	Declaration error
-84	Bad array size
-85	Structure fault
-86	Missing endp
-87	Syntax Error
-88	Mismatched (or)
-89	Bad field list
-90	Too complex
-91	Missing ,

-92	Variables too large
-93	Bad assignment
-94	Bad array index
-95	Inconsistent procedure arguments

OPL specific errors

-96	Illegal opcode (corrupt module - translate again)
-97	Wrong number of arguments (to a function or parameters to a procedure)
-98	Undefined externals (a variable has been encountered which hasn't been declared)
-99	Procedure not found
-100	Field not found
-101	File already open
-102	File not open
-103	Record too big (data file contains record too big for OPL)
-104	Module already loaded (when trying to LOADM)
-105	Maximum modules loaded (when trying to LOADM)
-106	Module does not exist (when trying to LOADM)
-107	Incompatible translator version (OPL file needs retranslation)
-108	Module not loaded (when trying to UNLOADM)
-109	Bad file type (data file header wrong or corrupt)
-110	Type violation (passing wrong type to parameter)
-111	Subscript or dimension error (out of range in array)
-112	String too long
-113	Device already open (when trying to LOPEN)
-114	Escape key pressed
-115	Incompatible runtime version
-116	ODB file(s) not closed
-117	Maximum drawables open (maximum 8 windows and/or bitmaps allowed)
-118	Drawable not open
-119	Invalid Window (window operation attempted on a bitmap)

11

Advanced Topics

This manual cannot cover every OPL command in detail, as some give access to the innermost workings of the Series 3. Some of these commands are touched on in this chapter.

The OS and CALL functions, described in the 'Alphabetic listing' chapter, give access to the complete range of Operating System interrupts.

Where files are kept

The internal memory and SSDs use a DOS-compatible directory structure, the same as that used on disks on business PCs. However, the Series 3 hides the complexities of this. You have only to supply a filename, and to say whereabouts a file is - internal, on an SSD, or on another computer to which the Series 3 is linked.

In fact, in order to completely specify a file, the Series 3 uses a *filing system*, *drive* (or *device*), *directory* and *filename*:

- The filing system usually specifies the computer, and is usually `LOC::` ('local' - the Series 3) or `REM::` ('remote' - an attached computer). This is always three letters and two colons.
- The drive is the area on that computer where the file is kept. On the Series 3 this can be `M:` (internal memory), `A:` (left SSD drive) or `B:` (right SSD drive).
- Every drive has one *root directory*, written as a backslash (\). This can "contain" files and/or other directories, each of which can contain more files and/or more directories. When you have "directories in directories" like this, they're often called *subdirectories*. Their names show where they are. For example, the root directory (\) could contain a directory called `\JO`, which might in turn contain a directory called `\JO\BACKUP`, which might contain some files.
- Filenames are composed of one to eight letters and/or numbers, optionally followed by a *file extension* comprised of a dot and from one to three letters/numbers. File extensions are by convention used to group different types of files. The Series 3 uses file extensions in this way, but hides this from you.

To completely specify a file, you add the four parts together. The directory part must end with a backslash. So an OPL module named `TEST`, in a directory called `\JO` in the Series 3 internal memory can be specified as `LOC::M:\JO\TEST.OPL`. If this file were in the `\JO\BACKUP` directory, it would be completely specified as `LOC::M:\JO\BACKUP\TEST.OPL`. If it were in the root directory, you would specify it as `LOC::M:\TEST.OPL`.

A full file specification may be up to 128 characters long.

In OPL, unless you say otherwise, files are kept on the Series 3 (`LOC::`), in the internal memory (`M:`). The directories and file extensions used are:

Type of file	Directory	File extension
OPL modules	\OPL	.OPL
translated modules	\OPO	.OPO
data files	\OPD	.ODB
bitmaps	\OPD	.PIC

Using file specifications in OPL

OPL commands which specify a filename, such as OPEN, CREATE, gLOADBIT and so on, can also use any or all of the other parts that make up a full file specification. (Normally, the only other part you might use is the drive name, if the file were on an SSD.) So for example, `OPEN "REM::C:\ADDR.TXT"...` tries to open a data file called `ADDR.TXT` on the root directory of a hard disk `C:` on an attached PC.

You can use the PARSE$ function if you need to build up complex filenames. See the alphabetic listing for more details of PARSE$.

Unless you have a good reason, though, it's best not to change directories or file extensions for files on the Series 3. You can lose information this way, unless you're careful.

Control of directories

Use the MKDIR command to make a new directory. For example, `MKDIR "M:\MINE\TEMP"` creates a `M:\MINE\TEMP` directory, also creating `M:\MINE` if it is not already there. An error is raised if the chosen directory exists already - use `TRAP MKDIR` to avoid this.

SETPATH sets the current directory for file access - for example, `SETPATH "A:\DOCS"`. LOADM continues to use the directory of the running program, but all other file access will be to the new directory.

Use RMDIR to remove a directory - for example, `RMDIR "M:\MINE"` removes the `MINE` directory on `M:\`. An error is raised if the directory does not exist - use `TRAP RMDIR` to avoid this.

You can only remove empty directories.

File specifications on REM::

You should not assume that remote file systems use DOS-like file specifications - for example, an Apple Macintosh specification is `disk:folder:folder:filename`. You can only assume that there will be four parts - disk/device, path, filename and (possibly null) extension. PARSE$, however, will always build up or break down REM:: file specifications correctly.

I/O functions and commands

OPL includes powerful facilities to handle input and output ('I/O'). These functions and commands provide access to all files and devices on the Series 3. However, only file I/O is covered in detail here, since a full description of device I/O is beyond the scope of this manual.

This chapter describes how to open, close, read and write to files, and how to set the position in a file. The data file handling commands and functions have been designed for use specifically with data files. The I/O functions and commands are designed for general file access. You don't need to use them to handle data files.

These are powerful functions and commands and they must be used with care. Before using them you must read this chapter closely and have a good grounding in OPL in general.

Error handling

You should have a good understanding of error handling before using the I/O functions.

The functions in this section never raise an OPL error message. Instead they return a value - if this is less than zero an error has occurred. It is the responsibility of the programmer to check all return values, and handle errors appropriately. Any error number returned will be one of those in the list given in the error handling chapter. You can use ERR$ to display the error as usual.

Notes

var

In this section, `var` denotes an argument which **must** be a LOCAL or GLOBAL variable, **because the function passes information back by setting these variables.** Single elements of arrays may also be used, but not field variables or procedure parameters. For example:

```
ret%=IOOPEN(var handle%,name$,mode%)
```

indicates that `IOOPEN(h%,"abc",0)` is OK while `IOOPEN(100,"abc",0)` is incorrect.

'`var`' is just to show you where you must use a suitable variable - you don't actually type it.

 # feature
In functions using 'var' the *address*, not the value, of the variable is passed. Fields, parameters and constants do not have an address which can be passed in this way. This means that you cannot use arguments whose value was passed from another procedure. This could be a problem, as you might, for example, want the code for a 'Safe Open' to be a routine.

You can get round this problem - pass the ADDR of the variable to the 'Safe Open' routine, and use a # at the front of the 'var' argument, like this:

```
sOpen:(ADDR(d%),"d.bin",$100)
```

and:

```
PROC sOpen:(handle%,name$,mode%)
   REM 'Safe Open'; handles errors
   ret%=IOOPEN(#handle%,name$,mode%)
   IF ret% :RAISE ret% :ENDIF
ENDP
```

means that the expression following it is treated as an address, not as a variable whose address is to be used. `IOOPEN(handle%...)` is equivalent to `IOOPEN(#(ADDR(handle%))...)`
The expression following # is treated as an integer expression. You can make use of # wherever 'var' is specified.

Handles
Many of these functions use a *handle*, which must be an integer variable. IOOPEN assigns this handle variable a value, which subsequent I/O functions use to access that particular file. Each file you IOOPEN needs a **different** handle variable.

Opening a file with IOOPEN

```
ret%=IOOPEN(var handle%,name$,mode%)
```
or
```
ret%=IOOPEN(var handle%,address%,mode%)
```
for unique file creation

Creates or opens a file (or device) called `name$` and sets `handle%` to the handle to be used by the other I/O functions.

`mode%` specifies how the file is to be opened. It is formed by ORing together values which fall into the three following categories:

Mode Category 1 - Open mode
One and only one of the following values must be chosen from this category.

$0000 Open an existing file (or device). The initial current position is set to the start of the file.

$0001 Create a file which must not already exist.

$0002 Replace a file (truncate it to zero length) or create it if it does not exist.

$0003 Open an existing file for appending. The initial current position is set to the end of the file. For text format files (see $0020 below) this is the only way to position to end of file.

$0004 Creates a file with a unique name. For this case, you must use the **address of a string** instead of `name$`. This string specifies only the path of the file to be created (any file name in the string is ignored). The string at `address%` is then set by IOOPEN to the unique file name generated (this will include the full path). The string must be large enough to take 130 characters (the maximum length file specification). For example:

```
s$="M:\home\"
IOOPEN(handle%,ADDR(s$),mode%)
PRINT s$ REM displays generated file name
```

This mode is typically used for temporary files which will later be deleted or renamed.

Mode Category 2 - File format
One and only one of the following values must be chosen from this category. When creating a file, this value specifies the format of the new file. When opening an existing file, make sure you use the format with which it was created.

$0000 The file is treated as a byte stream of binary data with no restriction on the value of any byte and no structure imposed upon the data. Up to 16K can be read from or written to the file in a single operation.

$0020 The file is treated as a sequence of variable length records. The records are assumed to contain text terminated by any combination of the CR and LF (`$0D`, `$0A`) characters. The maximum record length is 256 bytes and Control-Z (`$1A`) marks the end of the file.

Mode Category 3 - Access flags
Any combination of the following values may be chosen from this category.

$0100 Update flag. Allows the file to be written to as well as read. If not set, the file is opened for reading only. You **must** use this flag when creating or replacing a file.

$0200 Choose this value if you want the file to be open for *random access* (not sequential access), using the IOSEEK function.

$0400 Specifies that the file is being opened for sharing - for example, with other running programs. Use if you want to read, not write to the file. If the file is opened for writing (`$0100` above), this flag is ignored, since sharing is then not feasible. If not specified, the file is locked and may only be used by this running program.

Closing a file with IOCLOSE

Files should be closed when no longer being accessed. This releases memory and other resources back to the system.

```
ret%=IOCLOSE(handle%)
```

Closes a file (or device) with the handle `handle%` as set by IOOPEN.

Reading a file with IOREAD

`ret%=IOREAD(handle%,address%,maxLen%)`

Reads up to `maxLen%` bytes from a file with the handle `handle%` as set by IOOPEN. `address%` is the address of a buffer into which the data is read. This buffer must be large enough to hold a maximum of `maxLen%` bytes. The buffer could be an array or even a single integer as required. No more than 16K bytes can be read at a time.

The value returned to `ret%` is the actual number of bytes read or, if negative, is an error value.

Text files

If `maxLen%` exceeds the current record length, data only up to the end of the record is read into the buffer; no error is returned and the file position is set to the next record.

If a record is longer than `maxLen%`, the error value 'Record too large' (-43) is returned. In this case the data read is valid but is truncated to length `maxLen%`, and the file position is set to the next record.

A string array `buffer$(255)` could be used, but make sure that you pass the address ADDR`(buffer$)+1` to IOREAD. This leaves the leading byte free. You can then POKEB the leading byte with the count (returned to `ret%`) so that the string conforms to normal string format. See the example program.

Binary files

If you request more bytes than are left in the file, the number of bytes actually read (even zero) will be less than the number requested. So if `ret%<maxLen%`, end of file has been reached. No error is returned by IOREAD in this case, but the next IOREAD would return the error value 'End of file' (-36).

To read up to 16K bytes (8192 integers), you could declare an integer array `buffer%(8192)`.

Writing to a file

```
ret%=IOWRITE(handle%,address%,length%)
```

Writes `length%` bytes stored in a buffer at `address%` to a file with the handle `handle%`.

When a file is opened as a binary file, the data written by IOWRITE overwrites data at the current position.

When a file is opened as a text file, the data written by IOWRITE overwrites the current record and discards any following records. IOWRITE writes a single record; the closing CR/LF is automatically added.

Positioning within a file

```
ret%=IOSEEK(handle%,mode%,var offset&)
```

Seeks to a position in a file that has been opened for random access (see IOOPEN above).

`mode%` specifies how the argument `offset&` is to be used. `offset&` may be positive to move forwards or negative to move backwards. The values for `mode%` which are within the scope of this manual are defined as follows:

1 Set position in a binary file to the absolute value specified in `offset&`, with 0 for the first byte in the file.

2 Set position in a binary file to `offset&` bytes from the end of the file.

3 Set position in a binary file to `offset&` bytes relative to the current position.

6 Rewind a text file to the first record. `offset&` is not used, but you must still pass it as a parameter, for compatibility with the other cases.

IOSEEK sets the variable `offset&` to the absolute position set.

Example

This program opens a plain text file, such as one created with the 'Save as' option in the Word Processor or Database, and types it to the screen. Press Esc to quit and any other key to pause the typing to the screen.

```
PROC ioType:
   LOCAL ret%,fName$(128),txt$(255),address%
   LOCAL handle%,mode%,k%

   PRINT "Filename?", :INPUT fName$ :CLS
   mode%= $0400 OR $0020
REM open=$0000, text=$0020, share=$0400
   ret%=IOOPEN(handle%,fName$,mode%)
   IF ret%<0
      showErr:(ret%)
      RETURN
   ENDIF
   address%=ADDR(txt$)
   WHILE 1
      k%=KEY
      IF k% : REM if keypress
         IF k%=27 REM Esc pressed
            RETURN
         REM otherwise wait for a key
         ELSEIF GET=27
            RETURN REM Esc pressed
         ENDIF
      ENDIF
      ret%=IOREAD(handle%,address%+1,255)
      IF ret%<0
         IF ret%<>-36 REM NOT EOF
            showErr:(ret%)
         ENDIF
         BREAK
      ELSE
         POKEB address%,ret%
REM poke leading byte count
         PRINT txt$
      ENDIF
   ENDWH
   ret%=IOCLOSE(handle%)
   IF ret%
      showErr:(ret%)
   ENDIF
   PAUSE -100 :KEY
ENDP
```

```
PROC showErr:(val%)
   PRINT "Error",val%,err$(val%)
   GET
ENDP
```

I/O device handling and asynchronous requests

The following I/O functions provide access to devices. A full description is not within the scope of this manual, since these functions require extensive knowledge of the Series 3 operating system and related programming techniques. The syntax and argument descriptions are provided here for completeness.

More details can be found in the documentation for the Psion OPL and 'C' Development Systems.

`ret%=IOW(handle%,func%,var arg1,var arg2)`

The device driver opened with `handle%` (as returned by IOOPEN) performs the synchronous I/O function `func%` with the two further simple arguments. The size and structure of these two arguments is specified by the particular device driver's documentation.

`ret%=IOA(handle%,func%,var status%,var arg1,var arg2)`

The device driver opened with `handle%` (as returned by IOOPEN) performs the asynchronous I/O function `func%` with two further simple arguments. The size and structure of these two arguments is specified by the particular device driver's documentation.
Asynchronous means that the IOA returns immediately, and the OPL program can carry on with other statements. `status%` will usually be set to -46, which means that the function is still pending.
When, at some later time, the function completes, `status%` is automatically changed. (For this reason, `status%` should usually be global - if the program is still running, `status%` must be available when the request completes, or the program will probably crash.)
`status%` becomes:

>=0 function completed without error
<0 function completed with error. The error number is specific to the device driver

At the same time, a *signal* is sent to the running OPL program.
In most cases, you cannot perform another I/O read/write function to this device until you first read the signal of this function's completion. If this is the only I/O device with a function pending, wait for the signal with `IOWAITSTAT status%`. (If you have other functions pending on other devices, you must use IOWAIT and IOSIGNAL. These commands are described below.)
Alternatively, you can cancel the pending function with `IOW(handle%,4)`. The program will still receive a signal, which should be read with IOWAITSTAT or IOWAIT.
If an OPL program is ready to exit, it does not have to wait for any signals from pending IOA calls.

`IOWAIT`

Wait for an asynchronous request (such as one requested by IOA) to complete. IOWAIT returns when any asynchronous I/O function completes. Check `status%` to see whether it was the function which you called with IOA. You must keep a count of the number of times IOWAIT returns due to other functions completing. When `status%` finally shows that IOWAIT has returned because of your function completing, you must then call IOSIGNAL once for each other function which completed, to replace these other signals.
If you have no other functions pending on different I/O handles, use IOWAITSTAT instead.

`IOSIGNAL`

Replace a signal of an I/O function's completion. See IOWAIT.

`IOWAITSTAT` *`var status%`*

Wait for a particular asynchronous function, called with IOA, to complete. This saves using IOWAIT, checking each time to see if it was the desired function completing, and finally calling IOSIGNAL for each unexpected function completion.

`IOYIELD`

Ensure that any asynchronous wait-handler in a device is given a chance to run. Some devices are unable to perform an asynchronous request if an OPL program becomes computationally intensive, using no I/O (screen, keyboard etc) at all. In such cases, the OPL program should use IOYIELD before checking its `status%` variable. IOYIELD is the equivalent of

IOSIGNAL followed by IOWAIT - the IOWAIT returns immediately with the signal from IOSIGNAL, but the IOWAIT causes any asynchronous handlers to run.

`err%=KEYA(var status%,key%(1))`

This is an asynchronous keyboard read function. You must declare an integer array with two elements - here, `key%(1)` and `key%(2)` - to receive the keypress information. If a key is pressed, the information is returned in this way:

- `key%(1)` is assigned the character code of the key.
- The least significant byte of `key%(2)` takes the key modifier, in the same way as KMOD - 2 for Shift down, 4 for Control down and so on. KMOD cannot be used with KEYA.
- The most significant byte of `key%(2)` takes the count of keys pressed (0 or 1).

KEYA needs an IOWAIT in the same way as IOA.

KEYA has been included in OPL because the handle of the keyboard driver is unknown to the programmer. Otherwise it is equivalent to `IOA(keyhand%,status%,key%(1))`.

`err%=KEYC(var status%)`

Cancels a KEYA.

Some useful IOW functions

IOW has this specification:

```
ret%=IOW(handle%,func%,var arg1,var arg2)
```

Here are some uses:

```
LOCAL i%,a%(2)
   i%=17
   a%(1)=$4000+font%-1
   a%(2)=style%
   IOW(-2,7,i%,a%())
```

sets the font type for the text window. `font%` is as for gFONT. `style%` selects a style which changes the font size - `1` (bold) or `2` (double size). Use `0` to set normal font.

Warning: this causes the current screen (set with SCREEN) to be resized and repositioned, and some unexpected results may occur. It may help to clear the screen before this call.

```
LOCAL i%,a%
   i%=0
   a%=style%
   IOW(-2,7,i%,a%)
```

sets the text style. `style%` may be `0` (normal), `2` (inverse), `4` (underline) or `6` (inverse and underline).

```
LOCAL a%(6)
   a%(1)=x1% :a%(2)=y1%
   a%(3)=x2% :a%(4)=y2%
   IOW(-2,8,a%(),a%()) REM 2nd a% is ignored
```

reads the cursor position in the rectangle `x1%,y1%` (top left), `x2%,y2%` (bottom right), writing the x and y positions to `a%(5)` and `a%(6)` respectively. This returns `0,0`, not `1,1`, as the top left.

Set `x1%,y1%,x2%,y2%` to the screen top left and bottom right (set by SCREEN), to read the cursor position in the current screen.

```
LOCAL i%,a%(6)
   i%=2
   a%(1)=x1% :a%(2)=y1%
   a%(3)=x2% :a%(4)=y2%
   IOW(-2,7,i%,a%())
```

clears a rectangle at `x1%,y1%` (top left), `x2%,y2%` (bottom right). If `y2%` is one greater than `y1%`, this will clear part or all of a line.

Example of IOW screen functions

The final four procedures in this module call the four IOW screen functions described beforehand. The rest of the module lets you select the function and values to use. It uses the technique used in the 'Friendlier interaction' chapter of handling menus and hot-keys by calling procedures with string expressions.

```
PROC iotest:
GLOBAL x1%,x2%,y1%,y2%
LOCAL i%,h$(4),a$(5)
   x1%=2 :y1%=2
   x2%=25 :y2%=5 REM our test screensize
   SCREEN x2%-x1%,y2%-y1%,x1%,y1%
   AT 1,1
   PRINT "Text window IO test"
   PRINT "PSION-Esc quits"
   h$="FSCR" REM our hot-keys
   DO
     i%=GET
     IF i%=$122 REM Menu key
        mINIT
        mCARD "Set","Font",%F,"Style",%S,"Rect",%R
        mCARD "Sense","Cursor",%C
        i%=MENU
        IF i% AND INTF(LOC(h$,CHR$(i%)))
           a$="proc"+chr$(i%)
           @(a$):
        ENDIF
     ELSEIF i% AND $200 REM hot-key
        i%=(i%-$200) AND $FFDF REM Uppercase it
        i%=LOC(h$,CHR$(i%)) REM One of ours?
        IF i%
           a$="proc"+MID$(h$,i%,1)
           @(a$):
        ENDIF REM ignore other weird keypresses
     ELSE REM some other key, so return it
        PRINT CHR$(i%);
     ENDIF
   UNTIL 0
ENDP
```

```
PROC procf:
   LOCAL n%,i%,a%
   dINIT "Font for text window"
   dCHOICE n%,"Font","Normal,bold,digit"
   dCHOICE i%,"Bold","No,Yes"
   dCHOICE a%,"Double","No,Yes"
   IF DIALOG
      CLS
      iofont:(n%,(i%-1)+8*(a%-1))
   ENDIF
ENDP

PROC procs:
   LOCAL r%,u%
   dINIT "Style for text window"
   dCHOICE r%,"Inverse","No,Yes"
   dCHOICE u%,"Underline","No,Yes"
   IF DIALOG
      iostyle:((u%-1)*4+(r%-1)*2)
   ENDIF
ENDP

PROC procc:
   LOCAL a&
   a&=iocurs&:
   PRINT "x";1+(a& AND &ffff);
   PRINT "y";1+(a&/&10000);
ENDP

PROC procr:
   LOCAL xx1%,yy1%,xx2%,yy2%
   LOCAL xx1&,yy1&,xx2&,yy2&
   dINIT "Clear rectangle"
   dLONG xx1&,"Top left x",1,x2%-x1%
   dLONG yy1&,"Top left y",1,y2%-y1%
   dLONG xx2&,"Bottom left x",2,x2%-x1%
   dLONG yy2&,"Bottom left y",2,y2%-y1%
   IF DIALOG
      xx1%=xx1&-1 :xx2%=xx2&-1
      yy1%=yy1&-1 :yy2%=yy2&-1
      iorect:(xx1%,yy1%,xx2%,yy2%)
   ENDIF
ENDP
```

```
PROC iofont:(font%,style%)
   LOCAL i%,a%(6)
   i%=17 :REM font option
   a%(1)=$4000+font%-1
   a%(2)=style%
   IOW(-2,7,i%,a%())
ENDP

PROC iostyle:(style%)
   LOCAL i%,a%
   i%=0 :REM style option
   a%=style%
   IOW(-2,7,i%,a%)
ENDP

PROC iocurs&:
   LOCAL a%(4),a&
   REM don't change the order of these!
   a%(1)=x1% :a%(2)=y1%
   a%(3)=x2% :a%(4)=y2%
   IOW(-2,8,a%(),a%()) REM 2nd a% is ignored
   RETURN a&
ENDP

PROC iorect:(xx1%,yy1%,xx2%,yy2%)
   LOCAL i%,a%(6)
   i%=2 :REM clear rect option
   a%(1)=xx1% :a%(2)=yy1%
   a%(3)=xx2% :a%(4)=yy2%
   IOW(-2,7,i%,a%())
ENDP
```

IOA example - setting an alarm

The ALM: device provides access to alarms. When writing to it with IOW or IOA, you can use these two functions:

- function=1 - only the date (and no time) is shown on the screen when the alarm rings - e.g. `Thu 5 Sep`
- function=2 - the day and time are shown - e.g. `Thu 11:54`

In either case, you must pass these two arguments:

- An array of 2 long integers - the first is the time for the alarm to go off, and the second is the time for which it is due. Both are given in seconds since midnight a.m. on 1/1/1970.
- A message, as a **zero-terminated** string of up to 64 characters.

This procedure asks for the information for an alarm, and sets it (as type 2 - day and time to be shown when the alarm rings). If you press the Time button, and this is the next alarm to ring, it is shown as a 'RunOpl' alarm.

```
PROC alm:
LOCAL h%,a&(2),a$(64),b$(65),d&,t&,t2&,a%,r%,s%
r%=IOOPEN(h%,"ALM:",0)
IF r%<0 :RAISE r% :ENDIF
d&=DAYS(DAY,MONTH,YEAR) REM today
t&=DATETOSECS(1970,1,1,HOUR,MINUTE,0)
DINIT "Set alarm"
DTIME t&,"Time",0,0,DATETOSECS(1970,1,1,23,59,59)
DDATE d&,"Date",d&,DAYS(31,12,2049)
DTIME t2&,"Alarm advance time",2,0,86399
DEDIT a$,"Message"
IF DIALOG
   a&(2)=86400*(d&-25567)+t&
   a&(1)=a&(2)-t2&
   b$=a$+CHR$(0)
   a%=ADDR(b$)
   r%=IOA(h%,2,s%,a&(),#a%+1)
   IF r%<0 :RAISE r% :ENDIF
ENDIF
r%=IOCLOSE(h%)
IF r%<0 :RAISE r% :ENDIF
ENDP
```

Use this as a whole program itself - do not call it from another procedure.

The Series 3 allows up to 20 alarms to be pending. If you run this program more than once, you should always allow for four Time alarms, and one for each Agenda file you use.

OPL applications (OPAs)

You can make an OPL program appear as an icon in the system screen, and behave like the other icons there, by converting it into an *OPL application*, or *OPA*. There are five different types of OPA, called type 0 to type 4:

- `TYPE 0` (like Calc): The OPA uses no files.
- `TYPE 1`: Only one file is used. A type 1 OPA will look the same as a type 0. The only difference is that the type 1 is using a file, of the same name as the OPA.
- `TYPE 2` (like World): You can have more than one file, but only one can be in use (bold) at any time.

  When you pick a new file to use, its name becomes bold, and the one that was previously bold reverts to normal. **What has actually happened is that the running OPA has switched files - it has not** closed down, and no new copy of the OPA is run.
- `TYPE 3` (like Data, Word and Agenda): You can have more than one file, and any number may be in use (bold) at a given time.

  When you select a new file, one of the running OPAs normally switches to this file, as with type 2 OPAs. You can, however, with Shift-Enter, start a new OPA running just for this file.
- `TYPE 4` (like RunOpl): Many files can be used, and any number may be in use at a given time. When you select a new file, a new version of the OPA is always run, to use the new file.

Types 3 and 4 allow more than one file to be in use (i.e. have their names in bold). When this happens **a separate version of the OPA runs for each bold file.** With types 0, 1 and 2, only one version of the OPA can be running at any time.

Initially, the OPA's name appears beneath the icon. If you move onto this name and press Enter, file-based OPAs (types 1 to 4) will use a file of this name. Types 2, 3 and 4 allow you to create lists of files below the icon (with the 'New file' option). You use the file lists in the same way as the lists under the other icons in the System screen.

You can stop a running OPA by moving the cursor onto its bold name and pressing Delete. After a 'Confirm' dialog, the System screen tells the OPA to stop running.

Defining an OPA

To make an OPA, your OPL file should **begin with** the APP keyword, followed by a name for the OPA. The name should begin with a letter, and comprise of 1 to 8 letters and/or numbers. (Note that it does not have quote marks.) The APP line may be followed by any or all of the keywords PATH, EXT, ICON and TYPE. Finally, use ENDA, and then the first procedure may begin as in a normal OPL file. Here is an example of how an OPA might start:

```
APP Expenses
   TYPE 3
   PATH "\EXP"
   EXT "EXP"
   ICON "\OPD\EXPENSES.PIC"
ENDA
```

Here is another example:

```
APP Picture
   TYPE 1
ENDA
```

TYPE takes an integer parameter from 0 to 4. The various types of OPA are outlined earlier. If you don't specify the type, 0 is used.

PATH gives the directory to use for this OPA's files. If you do not use this, the normal \OPD directory will be used. The maximum length, including a final \, is 19 characters. Don't include any drive name in this path.

EXT gives the file extension of files used by this OPA. If you do not specify this, .ODB is used. Note that the files used by an OPA do not have to be data files, as the I/O commands give access to files of all kinds. EXT does not define the file type, just the file extension to use. However, **for simplicity's sake, examples in this section use data files.**

(PATH and EXT provide information for the System screen - they do not affect the program itself.)

ICON gives the name of the bitmap file to use as the icon for this OPA. If no file extension is given, .PIC is used. If you do not use ICON, the OPA is shown on the System screen with a standard OPA icon.

Running the OPA

Once you've translated the OPL file, return to the System screen and use Install on the App menu to install the OPA in the System screen. (You only need to do this once.) Once installed, file-based OPAs are shown with the list of available files, if any are found. Otherwise, the name used after the APP keyword appears below the icon.

(Note: the translated OPA is saved in a \APP directory. If you previously translated the module without the APP...ENDA at the start, the old translated version will still be listed under the RunOpl icon, and should be deleted.)

The first thing a file-based OPA should do is to get the filename to use, and check whether it is meant to create it or open it. CMD$(2) returns the full name of the file to use; CMD$(3) returns "C" for "Create" or "O" for "Open". All file-based OPAs (types 1 to 4) should handle both these cases; if a "Create" fails because the file exists already, or an "Open" fails because it does not, OPL raises the error, and the OPA finishes running. You can of course handle this with your own error code.

How the SERIES 3 talks to an OPA

When the Series 3 wants an OPA to exit or to switch files, it sends it a *System message*, in the form of an *event*. This would happen if you press Delete to stop a running OPA, or select a new file for a type 2 or 3 OPA.

TESTEVENT and GETEVENT check for certain events, including both keypresses and System messages. All types of OPA **must** use these keywords to check for **both** keypresses and System messages; keyboard commands such as GET, KEY and KEYA cause other events to be **discarded.**

If TESTEVENT returns 'True', an event has occurred, and can be read with GETEVENT. This takes one parameter, the name of an integer array - for example, GETEVENT a%(). The array should be at least 6 integers long. (This is to allow for future upgrades - you only need use the first two integers.)

If the event is a keypress:
a%(1) = keycode (as for GET)
a%(2) AND $00ff= modifier (as for KMOD)
a%(2)/256= auto-repeat count (ignored by GET; you can ignore it too)

For non-key events (a%(1) AND $400) will be non-zero. If the event is a System message to change files or quit, a%(1)=$404. You should then use GETCMD$ to find the action required.

GETCMD$ returns a string, whose first character is "C", "O" or "X". If it is "C" or "O", the rest of the string is a filename.

The first character, which you can extract with LEFT$(GETCMD$,1), has the following meaning:

"C" - close down the current file, and create the specified new file.

"O" - close down the current file, and open the specified existing file.

"X" - close down the current file (if any) and quit the OPA.

`MID$(GETCMD$,2,128)` is the easiest way to extract the filename.

You can only call GETCMD$ once for each event. You should do so as soon as possible after reading the event.

Note: events are ignored while you are using keywords which pause the execution of the program - GET, GET$, EDIT, INPUT, PAUSE, MENU and DIALOG. If you need to use these keywords, use LOCK ON / LOCK OFF around them.

Example OPAs

Here is a type 0 OPA, which just prints the keys you press. The keyboard procedure `getk%` returns the key pressed, as with GET, but jumps to a procedure `endit:` if a System message to close down is received. (Type 0 OPAs do not receive "change file" messages.)

`getk%` does not return events with values 256 ($100) or above, as they are not simple keypresses. This includes the non-typing keys like Menu ($100-$1FF), hot-keys ($200-$3FF), and non-key events ($400 and above).

```
APP myapp0
   TYPE 0
   ICON "\opd\me"
ENDA

PROC start:
   GLOBAL a%(6),k%
   STATUSWIN ON :SCREEN 30,9,1,1 :gSETWIN 0,0,179,79
   PRINT "Q to Quit"
   PRINT "or press Delete in"
   PRINT " the System screen"
   DO
      k%=getk%:
      PRINT CHR$(k%);
   UNTIL (k% AND $ffdf)=%Q
ENDP

PROC getk%:
   DO
      GETEVENT a%()
      IF a%(1)=$404
         IF LEFT$(GETCMD$,1)="X"
            endit:
         ENDIF
      ENDIF
   UNTIL a%(1)<256
   RETURN a%(1)
ENDP
```

```
PROC endit:
   STOP
ENDP
```

Here is a similar type 3 OPA. It does the same as the previous example, but System messages to change files cause the procedure `fset:` to be called. The relevant files are opened or created; the name of the file in use is shown in the status window.

```
APP myapp3
   TYPE 3
   ICON "\opd\me"
ENDA

PROC start:
   GLOBAL a%(6),k%,w$(128)
   STATUSWIN ON
   SCREEN 30,9,1,1
   gSETWIN 0,0,179,79
   w$=CMD$(2)
   fset:(CMD$(3))
   PRINT "Q to Quit"
   PRINT "or press Delete in"
   PRINT " the System screen"
   PRINT "or create/swap files in"
   PRINT " the System screen"
   DO
      k%=getk%:
      PRINT CHR$(k%);
   UNTIL (k% AND $ffdf)=%Q
ENDP

PROC getk%:
   LOCAL t$(1)
   DO
      GETEVENT a%()
      IF a%(1)=$404
         w$=GETCMD$
         t$=LEFT$(w$,1)
         w$=MID$(w$,2,128)
         IF t$="X"
            endit:
         ELSEIF t$="C" OR t$="O"
            TRAP CLOSE
            IF ERR
               CLS :PRINT ERR$(ERR)
               GET :CONTINUE
            ENDIF
            fset:(t$)
         ENDIF
      ENDIF
```

```
   UNTIL a%(1)<256
   RETURN a%(1)
ENDP

PROC fset:(t$)
   LOCAL p%(6)
   IF t$="C"
      TRAP CREATE w$,A,A$
   ELSEIF t$="O"
      TRAP OPEN w$,A,A$
   ENDIF
   IF ERR
      CLS :PRINT ERR$(ERR)
      GET :STOP
   ENDIF
   SETNAME w$
ENDP

PROC endit:
   STOP
ENDP
```

You should, as in both these examples, be precise in checking for the System message; if in future the GETCMD$ function were to use values other than "C", "O" or "X", these procedures would ignore them.

In either case, if you need to check the modifier keys for the returned keypress, use `a%(2) AND $00ff` instead of KMOD.

SETNAME extracts the main part of the filename from any file specification (even one that is not DOS-like), in the same way as PARSE$. Using SETNAME ensures that the correct name will be used in the file list in the System screen. If an OPA lets you change files with its own 'Open file' option, it should always use SETNAME to inform the System screen of the new file in use.

To be strict, whenever creating a file, an OPA should first use PARSE$ to find the disk and directory requested. If the directory is the default directory of the OPA, it should use `TRAP MKDIR` to ensure that the directory exists.

When an OPA cannot respond

The LOCK command marks an OPA as locked or unlocked. When an OPA is locked with `LOCK ON`, the System will not send it events to change files or quit. If, for example, you move onto the file list in the System screen and press Delete to try to stop that running OPA, a message will appear, indicating that the OPA cannot close down at that moment.

You should use LOCK ON if your OPA uses a keyword, such as EDIT, which pauses the execution of the program. You might also use it when the OPA is about to go busy for a considerable length of time, or at any other point where a clean exit is not possible. Do not forget to use LOCK OFF as soon as possible afterwards.

An OPA is initially unlocked.

Designing an icon

An OPA icon is read from the top left 24 by 24 pixels in a bitmap file. Here is a simple example program which creates a suitable bitmap:

```
PROC myicon:
   gCREATE(0,0,24,24,1)
   gBORDER $200
   gAT 3,14
   gPRINT "me!"
   gSAVEBIT "me"
ENDP
```

In the OPA itself use the ICON keyword, as explained previously, to give the name of the bitmap file to use - here, `ICON "\opd\me"`.

OPAs and the status window

If you use STATUSWIN ON to display the status window, it shows the OPA's own icon and the name used with the APP keyword.

Important: The permanent status window is **behind** all other OPL windows. In order to see it, you **must** use SCREEN and gSETWIN to reduce the size of the text and graphics windows. You should ensure that your program does not create windows over the top of it.

The name can be changed with the SETNAME command. In general, an OPA should use SETNAME whenever it changes files, or creates a new file.

Other TYPE options

You can add either of these two numbers to the value you use with TYPE:

- $8000 (-32768) stops the System screen's 'New file' option from working, as for the RunOpl icon (translated OPL modules).
- $4000 (16384) stops the System screen from closing the OPA, as for the Time icon.

Section 2

Reference

There are two chapters listing the functions and commands:

Overview of functions and commands - this summarises functions and commands together, grouped according to their purpose. Use this if you know what you'd like to do, but not which function or command will do it.

Alphabetic listing - use this if you already know what function or command you need to use.

12

Overview of functions and commands

A *function* returns a value, while a *command* does not. In the following chapters functions and commands are listed together. Many tasks are performed by using a combination of functions and commands - finding out certain information and then acting on it. For example, you might use the COUNT function to find out the number of records there are in one of your data files, and then use the PRINT command to display this on the screen.

Typing commands, functions and arguments

- Commands, functions and arguments **may be typed in any combination of UPPER and lower case** - e.g. `GloBal x%,y`.
- **To put more than one statement on a line,** separate them by a space followed by a colon - e.g.:

```
CLS :PRINT "hello" :GET
```

  Any commands may be strung together like this, and as many of them as you like - provided the total line length does not exceed 255 characters. The colon is optional before a REM statement.
- Where one space is allowed, any number of spaces is allowed - e.g. `CLS : PRINT "Press Esc"` is valid.
- **Functions may be used as arguments in other functions or commands** - e.g. `PRINT LEFT$(A$,3)` and `a=COS(ABS(x))` are OK.

'Syntax errors'

'Syntax errors' are those detected when you try to translate modules. The OPL translator will return you to the line where the first error occurs.

All other errors occur while you're running a program. See the chapter on error handling for details.

How commands are specified

Commands are specified like this:

```
COMMAND  argument(s)
```

where `arguments(s)` follow the command **after a space** and are **separated from each other by a comma.** `argument(s)` may include:

x	Numeric expression (e.g. `SIN(30)`), variable (e.g. `price` or `z`) or literal value (e.g. `78.9`)

x%	Integer expression (e.g. `3*567`), variable (e.g. `price%` or `price&`) or literal value (e.g. `-5676`)
x&	Long integer expression (e.g. `3*56799`), variable (e.g. `profit&`) or literal value (e.g. `-5676869`)
x$	String expression (e.g. `MID$(a$)`), variable (e.g. `price$`) or literal value (e.g. `"word"`)
variable	Variable, e.g. `string$, tax%, z, q&`
log	Logical file name (`A`, `B`, `C` or `D`)
field	Field name
filename	Filename. Full file specifications, which may include filing system, device, directory and file extension, may be used. See the Advanced Topics chapter for more information.

For example, `AT` *x%,y%* might be used like this: `AT 15,2`

How functions are specified

Functions are specified like this:

variable`=FUNCTION(`*argument(s)*`)`

where `variable` may be `f%` or `f&` for a function returning an integer or long integer result, `f` for a function returning a floating-point result, or `f$` for a function returning a string result.

The argument(s):

- follow the command immediately
- are enclosed in brackets ( )
- are separated from each other in the brackets by a comma
- may include variables, literal values or expressions of the appropriate kind - integer, long integer, floating-point or string.

E.g. `f$=LEFT$(g$,x%)` might be used like this: `LEFT$("abc",2)`

If you use the wrong type of number as an argument it will, where possible, be converted. For example, you can use an integer for a floating-point argument, or a long integer for an integer argument. If the conversion is not possible - for example, if you use a floating-point number for an integer argument and its value is outside the range of integers - an error will be produced and the program stopped.

Some functions, such as `GET`, have no arguments.

Summary, listed by task

Program control

Loops, branches, jumps

Repeat a set of instructions	DO...UNTIL WHILE...ENDW
Do either one set of instructions or another set, or another, or another...	IF...ENDIF
Go...	
...to a specified label	GOTO
...to one of a list of labels	VECTOR/ENDV
...to the end/start of a repeating set of instructions	BREAK CONTINUE
...back to the calling procedure	RETURN
End the program	STOP

Error handling

Raise an error	RAISE
Put an explanatory comment in your program	REM
Declare an error-handler	ONERR
Let the program continue after an error	TRAP
After an error, find out what the error was	ERR ERR$

Screen and keyboard control

Display a string to be edited and get a value from the keyboard	EDIT
Get a value from the keyboard	INPUT
Display text, numbers etc.	PRINT
Set screen update method	gUPDATE
Pause...	
...for a number of seconds	PAUSE
...until a key is pressed	{GET {GET$
Position or hide the cursor	{AT {CURSOR Off/On
Clear the text window	CLS
Sound the buzzer	BEEP
Set the size/position of the text window	SCREEN
Find out which key was pressed, if any	{KEY, KEY$, GET, {GET$
Find out what combination of modifiers was pressed	KMOD
Disable/enable stopping from a running program	ESCAPE Off/On
Turn the Series 3 off	OFF

Files

General file management

Copy a file	COPY
Delete or rename a file	DELETE RENAME
Check to see if a certain file exists	EXIST
Find out what files there are	DIR$

OPL modules

Load an OPL module file so you can use the procedures in it	LOADM
Remove a module from memory	UNLOADM

Data files

Create a new data file	CREATE
OPEN or CLOSE a data file	OPEN, OPENR, CLOSE
Use a different data file that has been opened	USE
Copy a data file, optionally appending to another data file, and removing deleted records	COMPRESS
Once a data file has been OPENed, you can:	
Make a new record	APPEND
Change a record	UPDATE
Search for those records which contain a certain string	FIND
Erase a record	ERASE
Move to a different record	FIRST, LAST, NEXT, BACK, POSITION
Count the records	COUNT
Find whether you're at the end of the file yet	EOF

Find the current record number	POS
Find the number of bytes used by the current record	RECSIZE

▮ *Managing directories*

Create directory	MKDIR
Set current directory	SETPATH
Remove directory	RMDIR

Memory

Declare variables	{GLOBAL {LOCAL
Find how much free memory there is on a device	SPACE

Printing

Specify a device or file to print to	LOPEN
Close the print device or file opened with LOPEN	LCLOSE
Print to a device or file	LPRINT

Numbers

▮ *Trigonometry*

COS, SIN, TAN, ACOS, ASIN, ATAN

Convert between degrees and radians	{RAD {DEG

▮ *Other functions*

Raise *e* to a power	EXP

Logarithms	LN LOG
Pi as a constant	PI
Square root	SQR
Use random numbers	RND RANDOMIZE

When you've got a list of numbers

Find the greatest or smallest value in the list	MAX MIN
Average the list	MEAN
Add up the list	SUM
Find the standard deviation or variance	STD VAR

Changing the format of numbers

Knock the minus sign off a number	ABS IABS
Take whole number, removing any fractional part	INT INTF
Convert...	
...an integer into floating-point	FLT
...an integer into a hexadecimal string	HEX$
...a number into a string	FIX$, GEN$ SCI$, NUM$
...a string into a number	EVAL VAL

Strings

Copy characters from a string	LEFT$, MID$, RIGHT$
Repeat a string	REPT$
Make a string upper or lower case	LOWER$ UPPER$
Find out...	
...how long a string is	LEN
...the character code of the first character of a string	ASC
...where a certain string is within a string	LOC
Convert...	
... a string of digits to a number	VAL
...a number to a string	FIX$, GEN$ SCI$, NUM$
Get the character with a certain character code	CHR$

Date and time

Find out the current date and time...	
...as a string	DATIM$
...just the current time	SECOND, MINUTE, HOUR
...just the current date	DAY, MONTH YEAR
Find out...	
...the number of days between two dates	DAYS
...what day of the week, or what week number, a certain date falls in	DOW WEEK

Express...	
...1-12 as the name of a month	MONTH$
...1-7 as a day of the week	DAYNAME$
Convert between time formats	DATETOSECS SECSTODATE

Graphics

Drawing commands

Set current position	gAT gMOVE
Draw a line	gLINEBY gLINETO
Draw a sequence of lines	gPOLY
Draw a rectangle	gBOX gBORDER
Fill a rectangle	gFILL
Invert a rectangle	gINVERT
Scroll a rectangle	gSCROLL
Get current position	gX gY
Display a running clock	gCLOCK

Displaying graphics text

Display a list of expressions	gPRINT
Display text in a cleared box	gPRINTB
Display text neatly clipped	gPRINTCLIP
Find width required by text	gTWIDTH
Display text underlined/highlighted	gXPRINT

Setting styles

Set font to use	gFONT
Set to user-defined fonts	gLOADFONT gUNLOADFONT
Set graphics to set / clear / invert points	gGMODE
Set text to set / clear / invert / replace points	gTMODE
Set text to bold / underline / inverse / double / mono / italic	gSTYLE

Windows and bitmaps

Create a new window	gCREATE
Set position and/or size of a window	gSETWIN
Set order to show windows	gORDER
Get order of a window	gRANK
Set window visible / invisible	gVISIBLE
Get screen position of a window	gORIGINX gORIGINY
Create a bitmap	gCREATEBIT
Load a bitmap from file	gLOADBIT
Clear a window / bitmap	gCLS
Save window / bitmap to bitmap file	gSAVEBIT
Close down a window / bitmap	gCLOSE
Set which window / bitmap to use	gUSE
Fill an area with repetitions of another window / bitmap	gPATT
Copy an area from one window / bitmap to another	gCOPY
Read data back from a window / bitmap	gPEEKLINE
Get ID number of a window / bitmap	gIDENTITY
Get size of a window / bitmap	gWIDTH gHEIGHT

Get status information about a window / bitmap and about the cursor	gINFO

Menus

Start a new set of menus	mINIT
Define a menu	mCARD
Display menus	MENU

Dialogs

Start a new dialog	dINIT
Position a dialog	dPOSITION
Define text for a dialog	dTEXT
Define an edit box for a dialog	dEDIT
Define a secret edit box for a dialog	dXEDIT
Define a filename edit box for a dialog	dFILE
Define a choice list for a dialog	dCHOICE
Define a numeric edit box for a dialog	dFLOAT dLONG
Define a date/time edit box for a dialog	dDATE dTIME
Define exit keys for a dialog	dBUTTONS
Display a dialog	DIALOG
Display a simple "alert" dialog	ALERT

Status Window

Display/hide status window	STATUSWIN
Set a program's name	SETNAME

Screen messages

Display information messages	GIPRINT
Display 'busy' messages	BUSY

Further use

Run machine code	USR USR$
Find out where a certain variable is in memory	ADDR
Store a value in a specific place in memory	POKE commands
Find out the value stored at a certain place in memory	PEEK commands
Open any type of file	IOOPEN
Read from a file opened with IOOPEN	IOREAD
Write to a file opened with IOOPEN	IOWRITE
Close a file opened with IOOPEN	IOCLOSE
Position within a file opened with IOOPEN	IOSEEK

Keywords whose full specification is beyond the scope of this manual

Call an operating system service	CALL OS
Perform an aysnchronous I/O function	IOA
Wait for completion of a function performed by IOA	IOWAIT IOWAITSTAT

Signal completion of an I/O function	IOSIGNAL
Ensure an asynchronous handler runs	IOYIELD
Perform a synchronous I/O function	IOW
Perfom an asynchronous keyboard read	KEYA
Cancel a KEYA	KEYC
Get command line information	CMD$ GETCMD$
Parse a full file specification	PARSE$
Check for system events	GETEVENT TESTEVENT
Mark an OPA as locked or unlocked	LOCK

13

Alphabetic listing

Note
The example programs in this chapter do not include full error handling code. This means that the programs have been kept short and easy to understand, but may fail if, for example, you enter the wrong type of value for a variable.

If you want to develop programs from these examples, it is recommended that you add some error handling code to them. An earlier chapter covers Error handling.

ABS

Usage:
```
a=ABS (x)
```

Returns the absolute value of a floating-point number - that is, without any +/- sign - for example `ABS (-10.099)` is `10.099`

If `x` is an integer, you won't get an error, but the result will be converted to floating-point - for example `ABS (-6)` is `6.0`. Use IABS to return the absolute value as a long integer.

ACOS

Usage:
```
a=ACOS (x)
```

Returns the arc cosine, or inverse cosine (COS^{-1}) of `x`.

`x` must be in the range -1 to +1. The number returned will be an angle in radians. To convert the angle to degrees, use the DEG function.

ADDR

Usage:
```
a%=ADDR (variable)
```

Returns the address at which `variable` is stored in memory.

The values of different types of variables are stored in bytes starting at `ADDR (variable)`. See PEEK for details.

ALERT

Usage - any of:
```
r%=ALERT (m1$,m2$,b1$,b2$,b3$)
r%=ALERT (m1$,m2$,b1$,b2$)
r%=ALERT (m1$,m2$,b1$)
r%=ALERT (m1$,m2$)
r%=ALERT (m1$)
```

Presents an *alert* - a simple dialog - with the messages and keys specified, and waits for a response. `m1$` is the message to be displayed on the first line, and `m2$` on the second line. If `m2$` is not supplied or if it is a null string, the second message line is left blank.

Up to three keys may be used. `b1$`, `b2$` and `b3$` are the strings (usually words) to use over the keys. `b1$` appears over an `Esc` key, `b2$` over `Enter`, and `b3$` over `Space`. This means you can have `Esc`, or `Esc` and `Enter`, or `Esc`, `Enter` and `Space` keys. If no key strings are supplied, the word `CONTINUE` is used above an `Esc` key.

The key number - 1 for `Esc`, 2 for `Enter` or 3 for `Space` - is returned.

APP

Usage:
```
APP name
...
ENDA
```

Begins definition of an OPA. `name` gives the name of the OPA.

See the Advanced Topics chapter for more details of OPAs.

APPEND

Usage:
`APPEND`

Adds a new record to the end of the current data file. The record which was current is unaffected. The new record, the last in the file, becomes the current record.

The record added is made from the current values of the field variables `A.field1$`, `A.field2$`, and so on, of the current data file. If a field has not been assigned a value, zero will be assigned to it if it is a numeric field, or a null string if it is a string field.

Example:
```
PROC add:
OPEN "address",A,f1$,f2$,f3$
PRINT "ADD NEW RECORD"
PRINT "Enter name:",
INPUT A.f1$
PRINT "Enter street:",
INPUT A.f2$
PRINT "Enter town:",
INPUT A.f3$
APPEND
CLOSE
ENDP
```

To overwrite the current record with new field values, use UPDATE.

ASC

Usage:
`a%=ASC(a$)`

Returns the character code of the first character of `a$`.

See the character set in the back of the User Guide for the character codes. Alternatively, use this statement:

`A%=%character` to find the code for `character` - eg `%X` for 'X'.

If a$ is a null string ("") ASC returns the value 0.

Example `A%=ASC("hello")` returns 104, the code for h.

ASIN

Usage:
`a=ASIN(x)`

Returns the arc sine, or inverse sine (SIN^{-1}) of x.

x must be in the range -1 to +1. The number returned will be an angle in radians. To convert the angle to degrees, use the DEG function.

AT

Usage:
`AT x%,y%`

Positions the cursor at x% characters across the text window and y% rows down. `AT 1,1` always moves to the top left corner of the window. Initially, the window is the full size of the screen (extending from 1,1 to 40,9), but you can change its size and position with the SCREEN command.

A common use of AT is to display strings at particular positions in the text window. For example:
`AT 5,2 :PRINT "message".`

- PRINT statements *without* an AT display at the left edge of the window on the line below the last PRINT statement (unless you use ',' or ';') and strings displayed at the top of the window eventually scroll

off as more strings are displayed at the bottom of the window.

- Strings always overwrite anything that is on the screen - they do not cause things below them on the screen to scroll down.

Example:

```
PROC records:
LOCAL k%
OPEN "clients",A,name$,tel$
DO
  CLS
  AT 1,7
  PRINT "Press a key to"
  PRINT "step to next record"
  PRINT "or Q to quit"
  AT 2,3 :PRINT A.name$
  AT 2,4 :PRINT A.tel$
  NEXT
  IF EOF
    AT 1,6 :PRINT "End of file"
    FIRST
  ENDIF
  k%=GET
UNTIL k%=%Q OR k%=%q
CLOSE
ENDP
```

ATAN

Usage:

```
a=ATAN(x)
```

Returns the arc tangent, or inverse tanget (TAN^{-1}) of x.

The number returned will be an angle in radians. To convert the angle to degrees, use the DEG function.

BACK

Usage:

```
BACK
```

Makes the previous record in the current data file the current record.

If the current record is the first record in the file, then the current record does not change.

BEEP

Usage:

```
BEEP time%,pitch%
```

Sounds the buzzer. The beep lasts for `time%/32` seconds - so for a beep a second long make `time%=32`, etc. The maximum is 3840 (2 minutes).

The pitch (frequency) of the beep is `512/(pitch%+1)` KHz.

`BEEP 5,300` gives a comfortably pitched beep.

If you make `time%` negative, BEEP first checks whether the sound system is in use (perhaps by another OPL program), and returns if it is. Otherwise, BEEP waits until the sound system is free.

Example, giving a scale from middle C:

```
PROC scale:
LOCAL freq,n%
REM n% relative to middle A
n%=3 REM start at middle C
WHILE n%<16
  freq=440*2**(n%/12.0)
  REM middle A = freq 440Hz
  BEEP 8,512000/freq-1.0
  n%=n%+1
  IF n%=4 OR n%=6 OR n%=9 OR
                  n%=11 OR n%=13
    n%=n%+1
  ENDIF
ENDWH
ENDP
```

(Note that the `IF` statement should all be on the same line).

Alternatively, sound the buzzer with this statement: `PRINT CHR$(7)`. This beeps at a fixed pitch for a fixed length of time.

BREAK

Usage:
```
BREAK
```

Makes a program performing a DO...UNTIL or WHILE...ENDWH loop exit the loop and immediately execute the line following the UNTIL or ENDWH statement.

Example:
```
DO
  ...
  ....
  IF a=b
    BREAK
  ENDIF
  ..
  ...
  ....
UNTIL a=b
x%=3
..
```

BUSY

Usage:
```
BUSY str$,c%,delay%
or
BUSY str$,c%
or
BUSY str$
or
BUSY OFF
```

`BUSY str$` displays `str$` in the bottom left of the screen, until `BUSY OFF` is called. Use this to indicate 'Busy' messages, usually when an OPL program is going to be unresponsive to keypresses for a while.

If `c%` is given, it controls the corner in which the message appears:

`c%`	*corner*
0	top left
1	bottom left (default)
2	top right
3	bottom right

`delay%` specifies a delay time (in half seconds) before the message should be shown. Use this to prevent 'busy' messages from continually appearing very briefly on the screen.

Only one message can be shown at a time. The string to display can be up to 19 characters long.

CALL

Usage:
```
e%=CALL(s%,bx%,cx%,dx%,si%,di%)
```

This function enables you to make operating system calls. Specification of each call is not within the scope of this manual.

The INT number itself is the least significant byte of `s%`. The AH value (the subfunction number) is the most significant byte of `s%`. The values of the other arguments are passed to the corresponding 8086 registers.

This function requires ***extensive*** knowledge of the Operating System and related programming techniques. The syntax of this command is included here for completeness.

CHR$

Usage:
```
a$=CHR$(x%)
```

Returns the character with character code `x%`.

You can use it to display characters not easily available from the keyboard

- for example, the instruction
`PRINT CHR$(174)` displays «.

For the full character set, see the User Guide.

CLOSE

Usage:
`CLOSE`

Closes the current file (that is, the one which has been OPENed and most recently USEd).

If you've used ERASE to remove some records, CLOSE recovers the memory used by the deleted records, provided it is held either in the internal memory or on a Ram SSD.

CLS

Usage:
`CLS`

Clears the contents of the text window.

The cursor then goes to the beginning of the top line. If you have used CURSOR OFF the cursor is still positioned there, but is not displayed.

CMD$

Usage:
`c$=CMD$(x%)`

Returns the command-line arguments passed when starting a program. Null strings may be returned. `x%` should be from 1 to 5. *cmd$(2)* to *cmd$(5)* are only for OPAs (OPL applications).

cmd$(1) returns the full path name used to start the running program.

cmd$(2) returns the full path name of the file to be used by an OPA application.

cmd$(3) returns `"C"` for "Create file" or `"O"` for "Open file". If the OPA is being run with a new filename, this will return `"C"`. This happens the very first time the OPA is used, and whenever a new filename is used to run it. Otherwise, the OPA is being run with the name of an existing file, and `cmd$(3)` will return `"O"`.

cmd$(4) returns the alias information, if any. Program aliases are beyond the scope of this manual.

cmd$(5) returns the application name, as declared with the APP keyword.

See the Advanced Topics chapter for more details of OPAs.

See also GETCMD$.

COMPRESS

Usage:
`COMPRESS src$,dest$`

Copies data file `src$` to another data file `dest$`. If `dest$` already exists, the records in `src$` are appended to the end of `dest$`.

Deleted records are not copied. This makes COMPRESS particularly useful when copying from a Flash SSD. (The space used by deleted records on a Ram SSD or in internal memory is automatically freed when you close the file.)

If you want `src$` to overwrite instead of append to `dest$`, use:
`TRAP DELETE dest$`
before the COMPRESS statement.

You can use wildcards if you wish to copy more than one file at a time. But if the first name contains any wildcards, the second name must not include a filename, just the device and directory to which the files are to be copied under their original names.

Example: to copy all the data files on A: (in \OPD, the default directory) to B:\BCK\:

```
COMPRESS "A:*.ODB","B:\BCK\"
```

(Remember the final backslash on the directory name.)

See COPY for copying any type of file.

CONTINUE

Usage:

```
CONTINUE
```

Makes a program immediately go to the UNTIL... line of a DO...UNTIL loop or the WHILE... line of a WHILE...ENDWH loop - i.e. to the test condition.

Example:

```
DO
  ...
  .....
  ....
  IF a<3.5
    CONTINUE
  ENDIF
  ..
  ....
  ...
UNTIL a=b
..
```

See also BREAK.

COPY

Usage:

```
COPY src$,dest$
```

Copies the file `src$`, which may be of any type, to the file `dest$`. Any existing file with the name `dest$` is deleted. You can copy across devices.

Use the appropriate file extensions to indicate the type of file, and wildcards if you wish to copy more than one file at a time:

- If `src$` contains wildcards, `dest$` must not specify a filename, just the device and directory to which the files are to be copied under their original names.
- You **must** specify either an extension or .* on the first filename. The file type extensions are listed in the Operating Manual.

Example:
To copy all the OPL files from internal memory (in \OPL) to B:\ME\:

```
COPY "M:\OPL\*.OPL","B:\ME\"
```

(Remember the final backslash on the directory name.)

See COMPRESS for more control over copying data files. If you use COPY to copy a data file, deleted records *are* copied and you cannot append to another data file.

There are more details of full file specifications in the Advanced Topics chapter.

COS

Usage:
`c=COS (x)`

Returns the cosine of x, where x is an angle in radians.

To convert from degrees to radians, use the RAD function.

COUNT

Usage:
`c%=COUNT`

Returns the number of records in the current data file.

This number will be 0 if the file is empty.

CREATE

Usage:
`CREATE file$,log,f1,f2,...`

Creates a data file called `file$`.

- The filename may be a full file specification of up to 128 characters. Field names may be up to 8 letters/numbers.
- The file may have up to 32 fields, as specified by `f1`, `f2`... (if viewed in the Database application, field `f1` starts on the top line of the window, `f2` is below it, etc.)
- `log` specifies the logical file name - A, B, C or D. This is used as an abbreviation for the file name when you use other data file commands such as USE.

Immediately after the CREATE statement, the file is open and can be accessed.

Example:

```
CREATE "CLIENTS",B,NAME$,PHONE$
```

would create a data file in the internal memory with the name `CLIENTS` and the logical name B.

CURSOR

Usage - one of the following:
```
CURSOR ON
CURSOR OFF
CURSOR id%,asc%,width%,height%
CURSOR id%
```

CURSOR ON switches the text cursor on at the current cursor position. Initially, no cursor is displayed.

You can switch on a graphics cursor in a window by following CURSOR with the ID of the window. This replaces any text cursor. At the same time, you can also specify the cursor's shape, and its position relative to the baseline of text.

`asc%` is the *ascent* - the number of pixels (-128 to 127) by which the top of the cursor should be above the baseline of the current font. `height%` and `width%` (both from 0 to 255) are the cursor's height and width.

If you do not specify them, the following default values are used:

`asc%`	=font ascent
`height%`	=font height
`width%`	=2

An error is raised if `id%` specifies a bitmap rather than a window.

CURSOR OFF switches off any cursor.

DATETOSECS

Usage:
`s&=DATETOSECS(yr%,mo%,dy%,hr%,mn%,sc%)`

Returns the number of seconds since 00:00 on 1 January 1970 at the date/time specified.

Raises an error for dates before 1 January 1970.

The value returned is an **unsigned** long integer. (Values up to +2,147,483,647, which is 03:14:07 on 19/1/2038, are returned as expected. Those from +2,147,483,648 upwards are returned as negative numbers, starting from -2,147,483,648 and increasing towards zero.)

See also SECSTODATE, HOUR, MINUTE, SECOND.

DATIM$

Usage:
`d$=DATIM$`

Returns the current date and time from the system clock as a string - for example:
`"Fri 16 Oct 1992 16:25:30"`
The string returned always has this format - 3 mixed-case characters for the day, then a space, then 2 digits for the day of the month, and so on.

DAY

Usage:
`d%=DAY`

Returns the current day of the month (1 to 31) from the system clock.

DAYNAME$

Usage:
`d$=DAYNAME$(x%)`

Converts x%,a number from 1 to 7, to the day of the week, expressed as a three letter string.

E.g. `d$=DAYNAME$(1)` returns MON.

Example:
```
PROC Birthday:
  LOCAL d&,m&,y&,dWk%
  DO
    dINIT
    dTEXT "","Date of birth",2
    dTEXT "","eg 23 12 1963",$202
    dLONG d&,"Day",1,31
    dLONG m&,"Month",1,12
    dLONG y&,"Year",1900,2155
    IF DIALOG=0 :BREAK :ENDIF
    dWk%=DOW(d&,m&,y&)
    CLS :PRINT DAYNAME$(dWk%),
    PRINT d&,m&,y&
    dINIT
    dTEXT "","Again?",$202
    dBUTTONS "No",%N,"Yes",%Y
  UNTIL DIALOG<>%y
ENDP
```

See also DOW.

DAYS

Usage:
`d&=DAYS(day%,month%,year%)`

Returns the number of days since 01/01/1900.

Use this to find out the number of days between two dates.

Example:

```
PROC deadline:
  LOCAL a%,b%,c%,deadlin&
  LOCAL today&,togo%
  PRINT "What day? (1-31)"
  INPUT a%
  PRINT "What month? (1-12)"
  INPUT b%
  PRINT "What year? (19??)"
  INPUT c%
  deadlin&=DAYS(a%,b%,1900+c%)
  today&=DAYS(DAY,MONTH,YEAR)
  togo%=deadlin&-today&
  PRINT togo%,"days to go"
  GET
ENDP
```

See also dDATE, SECSTODATE.

dBUTTONS

Usage - one of these:

```
dBUTTONS p1$,k1%,p2$,k2%,p3$,k3%
dBUTTONS p1$,k1%,p2$,k2%
dBUTTONS p1$,k1%
```

Defines *exit keys* to go at the bottom of a dialog.

From one to three exit keys may be defined. Each pair of `p$` and `k%` specifies an exit key; `p$` is the text to be displayed above it, while `k%` is the keycode of the key. DIALOG returns the keycode of the key pressed (in lower case for letters).

For alphabetic keys, use the `%` sign - `%A` means 'the code of `A`', and so on. An appendix lists the codes for keys (such as Tab) which are not part of the character set. If you use the code for one of these keys, its name (e.g. 'Tab', or 'Enter') will be shown in the key.

If you use a negative value for a `k%` parameter, that key is a 'Cancel' key. The corresponding positive value is used for the key to display and the value for DIALOG to return, but if you do press this key to exit, the *var* variables used in the commands like dEDIT, dTIME etc. will **not** be set.

The Esc key will always cancel a dialog box, with DIALOG returning 0. If you want to show the Esc key as one of the exit keys, use -27 as the `k%` parameter (its keycode is 27) so that the *var* variables will not be set if Esc is pressed.

There can be only one dBUTTONS item per dialog, and it takes up three lines on the screen. dBUTTONS must be the last dialog command you use before DIALOG itself.

Some keypresses, such as those using the Control key, cannot be specified.

This example presents a simple query, returning 'True' for Yes, or 'False' for No.

```
PROC query:
  dINIT
  dTEXT "","FORGET CHANGES",2
  dTEXT "","Are you sure?",$202
  dBUTTONS "No",%N,"Yes",%Y
  RETURN DIALOG=%y
ENDP
```

dCHOICE

Usage:

```
dCHOICE var choice%,p$,list$
```

Defines a choice list to go in a dialog.

`p$` will be displayed on the left side of the line. `list$` should contain the possible choices, separated by commas - for example, `"Yes,No"`. One of these will be displayed on the right side of the line, and ← and → can be used to move between the choices.

choice% must be a LOCAL or a GLOBAL variable. It specifies which choice should initially be shown - 1 for the first choice, 2 for the second, and so on. When you finish using the dialog, choice% is given a value indicating which choice was selected - again, 1 for the first choice, and so on.

dDATE

Usage:
dDATE *var lg&,p$,min&,max&*

Defines an edit box for a date, to go in a dialog.

p$ will be displayed on the left side of the line.

lg&, which must be a LOCAL or a GLOBAL variable, specifies the date to be shown initially. Although it will appear on the screen like a normal date, for example 15/03/92, lg& must be specified as "days since 1/1/1900".

min& and max& give the minimum and maximum values which are to be allowed. Again, these are in days since 1/1/1900. An error is raised if min& is higher than max&.

When you finish using the dialog, the date you entered is returned in lg&, in days since 1/1/1900.

The System setting determines whether years, months or days are displayed first.

See also DAYS, SECSTODATE.

dEDIT

Usage:
dEDIT *var str$,p$,len%*
or
dEDIT *var str$,p$*

Defines a string edit box, to go in a dialog.

p$ will be displayed on the left side of the line.

str$ is the string variable to edit. Its initial contents will appear in the dialog. The length used when str$ was defined is the maximum length you can type in.

len%, if supplied, gives the width of the edit box (allowing for widest possible character in the font). The string will scroll inside the edit box, if necessary. If len% is not supplied, the edit box is made wide enough for the maximum width str$ could possibly be.

DEG

Usage:
d=DEG(*x*)

Converts from radians to degrees.

Returns x, an angle in radians, as a number of degrees. The formula used is: 180*x/PI

All the trigonometric functions (SIN,COS etc.) work in radians, not degrees. You can use DEG to convert an angle returned by a trigonometric function back to degrees:

Example:

```
PROC xarctan:
  LOCAL arg,angle
  PRINT "Enter argument:";
  INPUT arg
  PRINT "ARCTAN of",arg,"is"
  angle=ATAN(arg)
  PRINT angle,"radians"
  PRINT DEG(angle),"degrees"
  GET
ENDP
```

To convert from degrees to radians, use RAD.

DELETE

Usage:
`DELETE filename$`

Deletes any type of file.

You can use wildcards - for example, to delete all the OPL files in B:\OPL
`DELETE "B:\OPL\*.OPL"`

The file type extensions are listed in the User Guide.

See also RMDIR.

dFILE

Usage:
`dFILE var str$,p$,f%`

Defines a filename edit box, to go in a dialog. A 'Disk' selector is automatically added on the line below.

`p$` will be displayed on the left side of the line.

`f%` controls the type of file editor, and the kind of input allowed. You can add together any of the following values:

value	*meaning*
1	use an edit box
2	allow directory names
4	directory names only
8	disallow existing files
16	query existing files
32	allow null string input

The first of the list is the most crucial. If you add 1 into `f%`, you will see an file edit box, as when creating a new file. If you do not add 1, you will see the 'matching file' selector, used when choosing an existing file.

If performing a 'copy to' operation, you might use 1+2+16, to specify a file edit box, in which you can type the name of a directory to copy to, and which will produce a query if you type the name of an existing file.

If asking for the name of a directory to remove, you might use 4, to allow an existing directory name only.

'Query existing' is ignored if 'disallow existing' is set. These two, as well as 'allow null string input', only work with file edit boxes, not 'matching file' selectors.

`str$` is the string variable to edit. Its initial contents always control the initial drive and directory used. For a file edit box, any filename part of `str$` is shown. For a 'matching file' selector, you can use wildcards in the filename part (such as `*.tmp`) to control which filenames are matched.

If `str$` does not contain any drive or directory information, the path as set by SETPATH is used. If SETPATH has not been used, the `\OPD` directory on the default drive (usually `M:`, 'Internal') is used.

If you use 8 with a matching file selector, as opposed to an edit box, it restricts the selection to files which match the filename/extension in `str$`.

You can always press Tab to produce the full file selector with a dFILE item.

`str$` must be declared to be at least 128 bytes long, or an error will be raised.

dFLOAT

Usage:
`dFLOAT var fp,p$,min,max`

Defines an edit box for a floating-point number, to go in a dialog.

`p$` will be displayed on the left side of the line.

`min` and `max` give the minimum and maximum values which are to be allowed. An error is raised if `min` is higher than `max`.

`fp` must be a LOCAL or a GLOBAL variable. It specifies the value to be shown initially. When you finish using the dialog, the value you entered is returned in `fp`.

DIALOG

Usage:
`n%=DIALOG`

Presents the dialog prepared by dINIT and commands such as dTEXT and dCHOICE. If you complete the dialog by pressing Enter, your settings are stored in the variables specified in dTEXT, dCHOICE etc., although you can prevent this with dBUTTONS.

If you used dBUTTONS when preparing the dialog, the keycode which ended the dialog is returned. Otherwise, DIALOG returns the line number of the item which was current when Enter was pressed. The top item (or the title line, if present), has line number 1.

If you cancel the dialog by pressing Esc, the variables are not changed, and 0 is returned.

dINIT

Usage:
`dINIT title$`
or
`dINIT`

Prepares for definition of a dialog, cancelling any existing one. Use dTEXT, dCHOICE etc. to define each item in the dialog, then DIALOG to display the dialog.

If `title$` is supplied it will be displayed at the top of the dialog, centred and with a line across the dialog below it.

DIR$

Usage:
`d$=DIR$(filespec$)` then
`d$=DIR$("")`

Lists filenames, including subdirectory names, matching a file specification. You can include wildcards in the file specification. If `filespec$` is just a directory name, include the final backslash on the end - for example, `"M:\TEMP\"` . Use the function like this:

- DIR$(*filespec$*) returns the name of the first file matching the file specification.
- DIR$("") then returns the name of the second file in the directory.
- DIR$("") again returns the third, and so on.
- When there are no more matching files in the directory, DIR$("") returns a null string.

Example, listing all the .DBF files in M:\DAT:

```
PROC dir:
  LOCAL d$(128)
  d$=DIR$("M:\DAT\*.DBF")
  WHILE d$<>""
    PRINT d$
    d$=DIR$("")
  ENDWH
  GET
ENDP
```

dLONG

Usage:

```
dLONG var lg&,p$,min&,max&
```

Defines an edit box for a long integer, to go in a dialog.

p$ will be displayed on the left side of the line.

min& and max& give the minimum and maximum values which are to be allowed. An error is raised if min& is higher than max&.

lg& must be a LOCAL or a GLOBAL variable. It specifies the value to be shown initially. When you finish using the dialog, the value you entered is returned in lg&.

DO...UNTIL

Usage:

```
DO
  statement
  statement
  .
  .
  .
UNTIL condition
```

DO forces the set of statements which follow it to execute repeatedly until the condition specified by UNTIL is met.

This is the easiest way to repeat an operation a certain number of times.

- Every DO must have its matching UNTIL to end the loop.
- If you set a condition which is never met, the program will go round and round, locked in the loop forever. You can escape by pressing PSION-Esc, provided you haven't set ESCAPE OFF. If you have set ESCAPE OFF, you will have to return to the System screen, move to the program name under the RunOpl icon, and press Delete.

DOW

Usage:

```
d%=DOW(day%,month%,year%)
```

Returns the day of the week - from 1 (Monday) to 7 (Sunday) - given the date.

day% must be between 1 and 31, month% from 1 to 12 and year% from 1900 to 2155.

For example, D%=DOW(4,7,1992) returns 6, meaning Saturday.

dPOSITION

Usage:
`dPOSITION x%,y%`

Positions a dialog. Use dPOSITION at any time between dINIT and DIALOG.

dPOSITION uses two integer values. The first specifies the horizontal position, and the second, the vertical. `dPOSITION -1,-1` positions to the top left of the screen; `dPOSITION 1,1` to the bottom right; `dPOSITION 0,0` to the centre, the usual position for dialogs.

`dPOSITION 1,0`, for example, positions to the right hand edge of the screen, and centres the dialog half way up the screen.

dTEXT

Usage:
`dTEXT p$,body$,t%`
or
`dTEXT p$,body$`

Defines a line of text to be displayed in a dialog.

`p$` will be displayed on the left side of the line, and `body$` on the right side. If you only want to display a single string, use a null string (`""`) for `p$`, and pass the desired string in `body$`. It will then have the whole width of the dialog to itself. An error is raised if `body$` is a null string.

`body$` is normally displayed left aligned (although usually in the right column). You can override this by specifying `t%`:

t%	effect
0	left align body$
1	right align body$
2	centre body$

In addition, you can add any or all of the following three values to `t%`, for these effects:

t%	effect
$100	use bold text for body$
$200	draw a line below this item
$400	(allow this item to be selected)

Only one line can be drawn across a dialog. It will be below the last item which asks for it, whether the title from dINIT or a dTEXT item.

dTIME

Usage:
`dTIME var lg&,p$,t%,min&,max&`

Defines an edit box for a time, to go in a dialog.

`p$` will be displayed on the left side of the line.

`lg&`, which must be a LOCAL or a GLOBAL variable, specifies the time to be shown initially. Although it will appear on the screen like a normal time, for example `18:27`, `lg&` must be specified as seconds after 00:00. A value of 60 means one minute past midnight; 3600 means one o'clock, and so on.

`min&` and `max&` give the minimum and maximum values which are to be allowed. Again, these are in seconds after 00:00. An error is raised if `min&` is higher than `max&`.

When you finish using the dialog, the time you entered is returned in `lg&`, in seconds after 00:00.

`t%` specifies the type of display required, as follows:

`t%`	*time display*
0	absolute time no seconds
1	absolute time with seconds
2	duration no seconds
3	duration with seconds

For example, `03:45` represents an absolute time while 3 hours 45 minutes represents a duration.

Absolute times are displayed in 24-hour or am/pm format according to the current setting in System.

dXINPUT

Usage:
`dXINPUT` *`var str$,p$`*

Defines a secret string edit box, such as for a password, to go in a dialog.

`p$` will be displayed on the left side of the line.

`str$` is the string variable to take the string you type. **Important:** `str$` must be at least eight characters long.

Initially the dialog does not show any characters for the string; the initial contents of `str$` are ignored. A special symbol will be displayed for each character you type, to preserve the secrecy of the string.

EDIT

Usage:
`EDIT` *`a$`*

Displays a string variable which you can edit directly on the screen. All the usual editing keys are available - the arrow keys move along the line, Esc clears the line, and so on.

When you have finished editing, press Enter to confirm the changes. If you press Enter before you have made any changes, then the string will be unaltered.

If you use EDIT in conjunction with a PRINT statement, use a comma at the end of the PRINT statement, so that the string to be edited appears on the same line as the displayed string:

```
...
PRINT "Edit address:",
EDIT A.address$
UPDATE
....
```

TRAP EDIT

If the Esc key is pressed while no text is on the input line, the 'Escape key pressed' error (number -114) will be returned by ERR - provided that the EDIT has been trapped. You can use this feature to enable someone to press the Esc key to escape from inputting a string.

See also INPUT, dEDIT.

ELSE(IF)/ENDA/ENDIF/ENDV/ENDWH

See IF/APP/VECTOR/WHILE

EOF

Usage:
```
e%=EOF
```

Finds out whether you're at the end of a file yet.

Returns -1 (true) if the end of the file has been reached, or 0 (false) if it hasn't.

When reading records from a file, you should test whether there are still records left to read, otherwise you may get an error.

Example:
```
PROC eoftest:
OPEN "myfile",A,a$,b%
DO
  PRINT A.a$
  PRINT A.b%
  NEXT
  PAUSE -40
UNTIL  EOF
PRINT "That's the last record"
GET
RETURN
ENDP
```

ERASE

Usage:
```
ERASE
```

Erases the current record in the current file.

The next record is then current. If the erased record was the last record in a file, then following this command the current record will be null and EOF will return true.

ERR

Usage:
```
e%=ERR
```

Returns the number of the last error which occurred, or 0 if there has been no error.

Example:
```
..
PRINT "Enter age in years"
age::
TRAP INPUT age%
IF ERR=-1
  PRINT "Number please:"
  GOTO age
ENDIF
...
```

See also ERR$.

See the Error handling chapter for full details, including the list of error numbers and messages.

ERR$

Usage:
```
e$=ERR$(x%)
```

Returns the error message for the specified error code x%.

ERR$(ERR) gives the message for the last error which occurred.

Example:
```
TRAP OPEN "B:\FILE",A,field1$
IF ERR
  PRINT ERR$(ERR)
  RETURN
ENDIF
```

See ERR.

See the Error handling chapter for full details, including the list of error numbers and messages.

ESCAPE OFF

Usage:
```
ESCAPE OFF...ESCAPE ON
```

ESCAPE OFF stops PSION-Esc being used to break out of the program when it is running. ESCAPE ON enables this feature again.

ESCAPE OFF takes effect only in the procedure in which it occurs, and in any sub-procedures that are called. PSION-Esc is always enabled when a program begins running.

If your program enters a loop which has no logical exit, and ESCAPE OFF has been used, you will have to return to the System screen, move to the program name under the RunOpl icon, and press the Delete key.

EVAL

Usage:
```
d=EVAL(s$)
```

Evaluates the mathematical string expression `s$` and returns the floating-point result. `s$` may include any mathematical function or operator (eg. `sin(x)/(2**3)`). Note that floating-point arithmetic is always performed.

For example:
```
DO
AT 10,5 :PRINT "Calc:",
TRAP INPUT n$
IF n$="" :CONTINUE :ENDIF
IF ERR=-114 :BREAK :ENDIF
CLS :AT 10,4
PRINT
"Result:",n$;"=";EVAL(n$)
UNTIL 0
```

See also VAL.

EXIST

Usage:
```
e%=EXIST(filename$)
```

Checks to see that a file exists.

Returns -1 ('True') if the file exists and 0 ('False') if it doesn't.

Use this function when creating a file to check that a file of the same name does not already exist, or when opening a file to check that it has already been created:
```
IF NOT EXIST("CLIENTS")
  CREATE "CLIENTS",A,names$
ELSE
  OPEN "CLIENTS",A,names$
ENDIF
...
```

EXP

Usage:
```
e=EXP(x)
```

Returns e^x - that is, the value of the arithmetic constant e (2.71828...) raised to the power of `x`.

EXT

Usage:
```
EXT name$
```

Gives the file extension of files used by an OPA.

This can only be used between APP and ENDA.

See the Advanced Topics chapter for more details of OPAs.

FIND

Usage:
```
f%=FIND(a$)
```

Searches the current data file for fields matching `a$`. The search starts from the current record, so use NEXT to progress to subsequent records. FIND makes the next record containing `a$` the current record and returns the number of the record found. Capitals and lower-case letters match.

You can use wildcards:

`?` matches any single character
`*` matches any group of characters.

So to find a record with a field containing both `Dr` and either BROWN or BRAUN, use:
```
F%=FIND("*DR*BR??N*")
```

`FIND("BROWN")` will find only those records with a field consisting solely of the string BROWN.

You can only search string fields.

FIRST

Usage:
```
FIRST
```

Positions to the first record in the current data file.

FIX$

Usage:
```
f$=FIX$(x,y%,z%)
```

Returns a string representation of the number `x`, to `y%` decimal places. The string will be up to `z%` characters long.

Example `FIX$(123.456,2,7)` returns `"123.46"`.

- If `z%` is negative then the string is right-justified - for example `FIX$(1,2,-6)` returns `"  1.00"` where there are two spaces to the left of the `1`.
- If `z%` is positive then no spaces are added - for example `FIX$(1,2,6)` returns `"1.00"`.
- If the number `x` will not fit in the width specified by `z%`, then the string will just be asterisks, for example `FIX$(256.99,2,4)` returns `"****"`.

See also GEN$, NUM$, SCI$.

FLT

Usage:
```
f=FLT(x&)
```

Converts an integer expression (either integer or long integer) into a floating-point number.

Example:
```
PROC gamma:(v)
  LOCAL c
  c=3E8
  RETURN 1/SQR(1-(v*v)/(c*c))
ENDP
```

You could call this procedure like this: `gamma:(FLT(a%))` if you wanted to pass it the value of an integer variable without having first to assign the integer value to a floating-point variable.

See also INT and INTF.

gAT

Usage:
```
gAT x%,y%
```

Sets the current position using absolute co-ordinates. `gAT 0,0` moves to the top left of the current drawable.

See also gMOVE.

gBORDER

Usage:
```
gBORDER flags%,width%,height%
```
or
```
gBORDER flags%
```

Draws a one-pixel wide border around the edge of the current drawable. If `width%` and `height%` are supplied, a border shape of this size is drawn with the top left corner at the current position. If they are not supplied, the border is drawn around the whole of the current drawable.

`flags%` controls three attributes of the border - a shadow to the right and beneath, a one-pixel gap all around, and the type of corners used:

`flags%`	*effect*
1	single pixel shadow
2	gap for single pixel shadow
3	double pixel shadow
4	gap for double pixel shadow
$100	one-pixel gap all round
$200	more rounded corners

You can combine the values to control the three different effects. For example, for rounded corners and a gap for a double pixel shadow, use `flags%=$204`.

Set `flags%=0` for no shadow, no gap, and sharper corners.

For example, to de-emphasise a previously emphasised border, use gBORDER with the shadow turned off:
```
gBORDER 3 REM show border
GET
gBORDER 4 REM border off
...
```

gBOX

Usage:
```
gBOX width%,height%
```

Draws a box from the current position, `width%` to the left and `height%` down. The current position is unaffected.

gCLOCK

Usage:
```
gCLOCK ON/OFF
```
or
```
gCLOCK ON,mode%,offset%
```
or
```
gCLOCK ON,mode%
```

Displays or removes a clock showing the system time. The current position in the current drawable is used. Only one clock may be displayed in each drawable.

`mode%` controls the type of clock:

`mode%`	*clock type*
1	small digital
2	medium, system setting
3	medium, analog
4	medium, digital
5	large, analog

You can OR the value with either or both of these:

$10 show date (small or medium)
$20 show seconds (small or large)
$40 show am/pm

`$31`, for example, specifies a small digital clock, showing both date and seconds. `$25` is a large analog clock with a moving second hand.

Digital clocks display in 24-hour or 12-hour mode according to the system-wide setting in Time. The 'am/pm' flag (`$40`) can be used with digital clocks in 12-hour mode, and with medium analog clocks.

`offset%` specifies an offset in minutes from the system time to the time displayed. This allows you to display a clock showing a time other than the system time.

If these parameters are not supplied, `mode%` is taken as 1, and `offset%` as 0.

gCLOSE

Usage:
`gCLOSE id%`

Closes the specified drawable that was previously opened by gCREATE, gCREATEBIT or gLOADBIT.

If the drawable closed was the current drawable, the default window (ID=1) becomes current.

An error is raised if you try to close the default window.

gCLS

Usage:
`gCLS`

Clears the whole of the current drawable and sets the current position to 0,0, its top left corner.

gCOPY

Usage:
`gCOPY id%,x%,y%,w%,h%,mode%`

Copies a rectangle of the specified size (width `w%`, height `h%`) from the point `x%,y%` in drawable `id%`, to the current position in the current drawable.

As this command can copy both set and clear pixels, the same modes are available as when displaying text. Set `mode%` = 0 for set, 1 for clear, 2 for invert or 3 for replace. 0, 1 and 2 act only on set pixels in the pattern; 3 copies the entire rectangle, with set and clear pixels.

The current position is not affected in either window.

gCREATE

Usage:
`id%=gCREATE(x%,y%,w%,h%,v%)`

Creates a window with specified position and size (width `w%`, height `h%`), and makes it both current and foreground. Sets the current position to 0,0, its top left corner. If `v%` is 1, the window will immediately be visible; if 0, it will be invisible.

Returns `id%` (2 to 8) which identifies this window for other keywords.

See also gCLOSE.

gCREATEBIT

Usage:
`id%=gCREATEBIT(width%,height%)`

Creates a bitmap with the specified size, and makes it the current drawable. Sets the current position to 0,0, its top left corner.

Returns `id%` (2 to 8) which identifies this bitmap for other keywords.

See also gCLOSE.

GEN$

Usage:
`g$=gen$(x,y%)`

Returns a string representation of the number `x`. The string will be up to `y%` characters long.

Example `GEN$(123.456,7)` returns `"123.456"` and `GEN$(243,5)` returns `"243"`

- If `y%` is negative then the string is right-justified - for example `GEN$(1,-6)` returns `"     1"` where there are five spaces to the left of the `1`.
- If `y%` is positive then no spaces are added - for example `GEN$(1,6)` returns `"1"`.
- If the number `x` will not fit in the width specified by `y%`, then the string will just be asterisks, for example `GEN$(256.99,4)` returns `"****"`.

See also FIX$, NUM$, SCI$.

GET

Usage:
`g%=GET`

Waits for a key to be pressed and returns the character code for that key.

For example, if the 'A' key is pressed with Caps Lock off, the integer returned is 97 (a), or 65 (A) if 'A' was pressed with the Shift key down.

The character codes of special keys, such as PAGE DOWN, are given in an appendix at the back of this manual.

You can use KMOD to check whether *modifier keys* (Shift, Control, PSION and Caps Lock) were used.

See the User Guide for the full character set.

See also KEY.

GET$

Usage:
`g$=GET$`

Waits until a key is pressed and then returns which key was pressed, as a string.

For example, if the 'A' key is pressed in lower case mode, the string returned is `"a"`.

You can use KMOD to check whether any *modifier keys* (Shift, Control, PSION and Caps Lock) were used.

See also KEY$.

GETCMD$

Usage:
```
w$=GETCMD$
```

Returns new command-line arguments to an OPA, after a "change files" or "quit" event has occurred. The first character of the returned string is "C", "O" or "X". If it is "C" or "O", the rest of the string is a filename.

The first character has the following meaning:

"C" - close down the current file, and create the specified new file.

"O" - close down the current file, and open the specified existing file.

"X" - close down the current file (if any) and quit the OPA.

You can only call GETCMD$ once for each System message.

See the Advanced Topics chapter for more details of OPAs.

See also CMD$.

GETEVENT

Usage:
```
GETEVENT var a%()
```

Waits for an event to occur. Returns with `a%()` specifying the event. The data returned in `a%()` depends on the type of event that occurred. If the event is a key-press, `(a%(1) AND $400)` is guaranteed to be zero. For other events `(a%(1) AND $400)` is guaranteed to be non-zero.

If a key has been pressed:
`a%(1)` = keycode (as for GET)
`a%(2) AND $00ff`= modifier (as for KMOD)
`a%(2)/256`= auto-repeat count (ignored by GET)

If a program has moved to foreground:
`a%(1)` = $401

If a program has moved to background:
`a%(1)` = $402

If the machine has switched on:
`a%(1)` = $403

If the Series 3 wants an OPA to change files or exit:
`a%(1)` = $404

Note: events are ignored while you are using keywords which wait for keypresses - GET, GET$, EDIT, INPUT, MENU and DIALOG. If you need to use these keywords in OPAs, use `LOCK ON / LOCK OFF` around them.

If you do use GETEVENT you should allow for other events to be specified in the future.

For a key-press event, the modifier is returned in `a%(2)` and is not returned by KMOD.

Note: If a non-key event such as 'foreground' occurs while a keyboard keyword such as GET, INPUT, MENU or DIALOG is being used, the event is discarded. So GETEVENT must be used if non-key events are to be monitored. (OPAs can still handle the $404 event correctly - see the LOCK command for more details.)

The array (or string of integers) **must** be at least 6 integers long.

See also TESTEVENT, GETCMD$.

gFILL

Usage:
`gFILL width%,height%,gMode%`

Fills a rectangle of the specified size from the current position, according to the graphics mode specified.

The current position is unaffected.

gFONT

Usage:
`gFONT fontId%`

Sets the font for current drawable to `fontId%`. The font may be one of the predefined fonts in the ROM or a user-defined font.

User-defined fonts must first be loaded by gLOADFONT, which returns the `fontId%` needed for gFONT.

See also gLOADFONT.

gGMODE

Usage:
`gGMODE mode%`

Sets the effect of all subsequent drawing commands - gLINEBY, gBOX etc. - on the current drawable.

mode%	*Pixels will be:*
0	set
1	cleared
2	inverted

When you first use drawing commands on a drawable, they set pixels in the drawable. Use gGMODE to change this. For example, if you have drawn a black background, you can draw a white box outline inside it with either `gMODE 1` or `gGMODE 2`, followed by gBOX.

gHEIGHT

Usage:
`height% = gHEIGHT`

Returns the height of the current drawable.

gIDENTITY

Usage:
`id%=gIDENTITY`

Returns the ID of the current drawable.

The default window has ID=1.

gINFO

Usage:
`gINFO var i%()`

Gets general information about the current drawable and about the graphics cursor (whichever window it may be in). The information is returned in the array `i%()` which must be at least 32 integers long.

The information is about the drawable in its current state, so e.g. the font information is for the current font in the current style.

The following information is returned:

`i%(1)`	lowest character code
`i%(2)`	highest character code
`i%(3)`	height of font
`i%(4)`	descent of font
`i%(5)`	ascent of font
`i%(6)`	width of '0' character
`i%(7)`	maximum character width
`i%(8)`	flags for font (see below)

i%(9-17)	name of font
i%(18)	current graphics mode (gGMODE)
i%(19)	current text mode (gTMODE)
i%(20)	current style (gSTYLE)
i%(21)	cursor state (ON=1,OFF=0)
i%(22)	ID of window containing cursor (-1 for text cursor)
i%(23)	cursor width
i%(24)	cursor height
i%(25)	cursor ascent
i%(26)	cursor x position in window
i%(27)	cursor y position in window
i%(28)	1 if drawable is a bitmap
i%(29-32)	reserved

i%(8) specifies a combination of the following font characteristics:

Value:	*Meaning:*
1	font uses standard ASCII character set
2	font uses Code Page 850 character set
4	font looks bolded
8	font looks italic
16	font has serifs
32	font is monospaced

Use PEEK$(ADDR(i%(9))) to read the name of the font as a string.

If the cursor is on (i%(21)=1), it is visible in the window identified by i%(22).

If the cursor is off (i%(21)=0), i%(22) to i%(27) should be ignored.

gINVERT

Usage:
gINVERT *width%,height%*

Inverts the rectangle width% to the right and height% down from the cursor position, except for the four corner pixels.

GIPRINT

Usage:
GIPRINT *str$,c%*
or
GIPRINT *str$*

GIPRINT displays an information message for about two seconds, in the bottom right corner of the screen. For example, GIPRINT "Not Found" displays Not Found. The string you specify can be up to 64 characters. If a string is too long for the screen, it will be clipped.

If c% is given, it controls the corner in which the message appears:

c%	*corner*
0	top left
1	bottom left
2	top right
3	bottom right (default)

Only one message can be shown at a time. You can make the message go away - for example, if a key has been pressed - with GIPRINT "".

gLINEBY

Usage:
gLINEBY *dx%,dy%*

Draws a line from the current position to a point dx% to the right and dy% down. Negative dx% and dy% mean

left and up respectively. The current position moves to the end of the line drawn.

`gLINEBY 0,0` sets the pixel at the current position.

See also gLINETO, gPOLY.

gLINETO

Usage:
`gLINETO x%,y%`

Draws a line from the current position to the point `x%,y%`. The current position moves to `x%,y%`.

To plot a single point, use gLINETO to the current position (or `gLINEBY 0,0`).

See also gLINEBY, gPOLY.

gLOADBIT

Usage:
`id%=gLOADBIT(name$,write%,i%)`
or
`id%=gLOADBIT(name$,write%)`
or
`id%=gLOADBIT(name$)`

Loads a bitmap from the named bitmap file and makes it the current drawable. Sets the current position to 0,0, its top left corner. If `name$` has no file extension `.PIC` is used. The bitmap is kept as a local copy in memory.

Returns `id%` (2 to 8) which identifies this bitmap for other keywords.

`write%=0` sets read-only access. Attempts to write to the bitmap in memory will be ignored, but the bitmap can be used by other programs without using more memory. `write%=1` allows you to write to and re-save the bitmap. This is the default case.

For bitmap files which contain more than one bitmap, `i%` specifies which one to load. For the first bitmap, use `i%=0`. This is also the default value. Bitmap files saved with gSAVEBIT have only one bitmap, and this parameter is not needed for them.

See also gCLOSE.

gLOADFONT

Usage:
`fontId%=gLOADFONT(name$)`

Loads the user-defined font `name$`. It returns a font ID; use this with gFONT to make the current drawable use this font. If `name$` does not contain a file extension, `.FON` is used.

gFONT itself is very efficient, so you should normally load all required fonts at the start of a program.

Note that the built-in Series 3 fonts are automatically available, and do not need loading.

See also gUNLOADFONT.

GLOBAL

Usage:
`GLOBAL variables`

Declares variables to be used in the current procedure (as does the LOCAL command) *and* (unlike LOCAL) in any procedures called by the current procedure, or procedures called by them.

The variables may be of 4 types, depending on the symbol they end with:

- Variable names not ending with `$ % & or ()` are *floating-point variables*, for example `price, x`
- Those ending with a `%` are *integer variables*, for example `x%, sales92%`
- Those ending with an `&` are *long integer variables*, for example `x&, sales92&.`
- Those ending with a `$` are *string variables*. String variable names must be followed by the maximum length of the string in brackets - for example `names$(12), a$(3)`

Array variables have a number immediately following them in brackets which specifies the number of elements in the array. Array variables may be of any type, for example:

```
x(6),y%(5),f$(5,12),z&(3)
```

When declaring string arrays, you must give two numbers in the brackets. The first declares the number of elements, the second declares their maximum length. For example `surname$(5,8)` declares five elements, each up to 8 characters long.

Variable names may be any combination of up to 8 numbers and alphabetic letters. They must start with a letter. The length includes the `% & or $` sign, but not the `()` in string and array variables.

More than one GLOBAL or LOCAL statement may be used, but they must be on separate lines, immediately after the procedure name.

See also LOCAL.

gMOVE

Usage:
`gMOVE dx%,dy%`

Moves the current position `dx%` to the right and `dy%` downwards, in the current drawable.

A negative `dx%` causes movement to the left; a negative `dy%` causes upward movement.

See also gAT.

gORDER

Usage:
`gORDER id%,position%`

Sets the window specified by `id%` to the selected foreground/background position, and redraws the screen. Position 1 is the foreground window, position 2 is next, and so on. Any position greater than the number of windows is interpreted as the end of the list.

On creation, a window is at position 1 in the list.

Raises an error if `id%` is a bitmap.

See also gRANK.

gORIGINX

Usage:
```
x%=gORIGINX
```

Returns the gap between the left side of the screen and the left side of the current window.

Raises an error if the current drawable is a bitmap.

gORIGINY

Usage:
```
y%=gORIGINY
```

Returns the gap between the top of the screen and the top of the current window.

Raises an error if the current drawable is a bitmap.

GOTO

Usage:
```
GOTO label or GOTO label::
..
..
label::
```

Goes to the line following the `label::` and continues from there. The label -

- Must be in the current procedure
- Must start with a letter and end with a double colon, although the double colon is not necessary in the GOTO statement
- May be up to 8 characters long excluding the colons.

gPATT

Usage:
```
gPATT id%,width%,height%,mode%
```

Fills a rectangle of the specified size from the current position with repetitions of the drawable `id%`.

As with gCOPY, this command can copy both set and clear pixels, so the same modes are available as when displaying text. Set `mode%` = 0 for set, 1 for clear, 2 for invert or 3 for replace. 0, 1 and 2 act only on set pixels in the pattern; 3 copies the entire rectangle, with set and clear pixels.

If you set `id%=-1` a pre-defined grey pattern is used.

The current position is unaffected.

gPEEKLINE

Usage:
```
gPEEKLINE id%,x%,y%,d%(),ln%
```

Reads a horizontal line from the drawable `id%`, length `ln%`, starting at `x%,y%`. The leftmost 16 pixels are read into `d%(1)`, with the first pixel read into the least significant bit.

The array `d%()` must be long enough to hold the data. You can work out the number of integers required with `((ln%+15)/16)` (using whole-number division).

gPOLY

Usage:
```
gPOLY a%()
```

Draws a sequence of lines, as if by gLINEBY and gMOVE commands.

The array is set up as follows:

a%(1)	starting x position
a%(2)	starting y position
a%(3)	number of pairs of offsets
a%(4)	dx1%
a%(5)	dy1%
a%(6)	dx2%
a%(7)	dy2% etc.

a%() contains pairs of offsets. Each pair is an amount dx%, dy% to move by. You should multiply each dx% by 2; if you are drawing a line by this dx%, dy% this number is correct, but if you are moving by this amount, add 1 to it.

Positive dx% means to the right, and positive dy% means down.

The current position is left at the start x, y position.

gPOLY is quicker than combinations of gAT, gLINEBY and gMOVE.

Example, to draw three horizontal lines 50 pixels long at positions 20,10, 20,30 and 20,50:

```
a%(1)=20 :a%(2)=10 REM 20,10
a%(3)=5 REM 5 operations
REM draw right 50
a%(4)=50*2   :a%(5)=0
REM move down 20
a%(6)=0*2+1  :a%(7)=20
REM draw left 50
a%(8)=-50*2  :a%(9)=0
REM move down 20
a%(10)=0*2+1 :a%(11)=20
REM draw right 50
a%(12)=50*2  :a%(13)=0
gPOLY a%()
```

gPRINT

Usage:
`gPRINT list of expressions`

Displays a list of expressions at the current position in the current drawable. All variable types are formatted as for PRINT.

Unlike PRINT, gPRINT does not end by moving to a new line. A comma between expressions is still displayed as a space, but a semi-colon has no effect. gPRINT without a list of expressions does nothing.

See also gPRINTB, gPRINTCLIP, gTWIDTH, gXPRINT.

gPRINTB

Usage - any of:

```
gPRINTB t$,w%,al%,tp%,bt%,m%
gPRINTB t$,w%,al%,tp%,bt%
gPRINTB t$,w%,al%,tp%
gPRINTB t$,w%,al%
gPRINTB t$,w%
```

Displays text t$ in a cleared box of width w% pixels. The current position is used for the left side of the box and for the baseline of the text.

al% controls the alignment of the text in the box - 1 for right aligned, 2 for left aligned, or 3 for centred.

tp% and bt% are the clearances between the text and the top/bottom of the box. Together with the current font size, they control the height of the box. An error is raised if tp% plus the font ascent is greater than 255.

m% controls the margins. For left alignment, m% is an offset from the left of the box to the start of the text.

For right alignment, m% is an offset from the right of the box to the end of the text. For centering, m% is an offset from the left or right of the box to the region in which to centre, with positive m% meaning left and negative meaning right.

If values are not supplied for some parameters, these defaults are used:

```
al%  left
tp%  0
bt%  0
m%   0
```

See also gPRINT, gPRINTCLIP, gTWIDTH, gXPRINT.

gPRINTCLIP

Usage:
```
w%=gPRINTCLIP(text$,width%)
```

Displays text$ at the current position, displaying only as many characters as will fit inside width% pixels. Returns the number of characters displayed.

See also gPRINT, gPRINTB, gTWIDTH, gXPRINT.

gRANK

Usage:
```
rank%=gRANK
```

Returns the foreground/background position, from 1 to 8, of the current window.

Raises an error if the current drawable is a bitmap.

See also gORDER.

gSAVEBIT

Usage:
```
gSAVEBIT name$,width%,height%
```
or
```
gSAVEBIT name$
```

Saves the current drawable as the named bitmap file. If width% and height% are given, then only the rectangle of that size from the current position is copied. If name$ has no file extension .PIC is used.

gSCROLL

Usage:
```
gSCROLL dx%,dy%,x%,y%,wd%,ht%
```
or
```
gSCROLL dx%,dy%
```

Scrolls pixels in the current drawable by offset dx%, dy%. Positive dx% means to the right, and positive dy% means down. The drawable itself does not change its position.

If you specify a rectangle in the current drawable, at x%, y% and of size wd%, ht%, only this rectangle is scrolled.

The areas dx% wide and dy% deep which are "left behind" by the scroll are cleared.

The current position is not affected.

gSETWIN

Usage:
```
gSETWIN x%,y%,width%,height%
```
or
```
gSETWIN x%,y%
```

Changes position and, optionally, the size of the current window.

An error is raised if the current drawable is a bitmap.

The current position is unaffected.

If you use this command on the default window, you must also use the SCREEN command to ensure that the area for PRINT commands to use is wholly contained within the default window.

gSTYLE

Usage:
`gSTYLE style%`

Sets the style of text displayed in subsequent gPRINT, gPRINTB and gPRINTCLIP commands on the current drawable.

`style%`	*Text style:*
0	normal
1	bold
2	underlined
4	inverse
8	double height
16	mono
32	italic

You can combine these styles by adding their values - for example, to set bold, underlined and double height, use `gSTYLE 11`, as 11=1+2+8.

This command does not affect non-graphics commands, like PRINT.

gTMODE

Usage:
`gTMODE mode%`

Sets the way characters are displayed by subsequent gPRINT, gPRINTB and gPRINTCLIP commands on the current drawable.

`mode%`	*Pixels will be:*
0	set
1	cleared
2	inverted
3	replaced

When you first use "graphics text" commands on a drawable, each dot in a letter causes a pixel to be set in the drawable. This is `mode%`=0.

When `mode%` is 1 or 2, graphics text commands work in a similar way, but the pixels are cleared or inverted. When `mode%` is 3, entire character boxes are drawn on the screen - pixels are set in the letter and cleared in the background box.

This command does not affect non-graphics commands, like PRINT.

gTWIDTH

Usage:
`width%=gTWIDTH(text$)`

Returns the width of `text$` in the current font and style.

See also gPRINT, gPRINTB, gPRINTCLIP, gXPRINT.

gUNLOADFONT

Usage:
`gUNLOADFONT fontId%`

Unloads a user-defined font that was previously loaded using gLOADFONT. Raises an error if the font has not been loaded.

The built-in Series 3 fonts are not held in memory and cannot be unloaded.

See also gLOADFONT.

gUPDATE

Usage - one of:
```
gUPDATE ON
gUPDATE OFF
gUPDATE
```

The Series 3 screen is usually updated whenever you display anything on it. gUPDATE OFF switches off this feature. The screen will be updated as few times as possible, although you can force an update by using the gUPDATE command on its own.

This can result in a considerable speed improvement in some cases. You might, for example, use gUPDATE OFF, then a sequence of graphics commands, followed by gUPDATE. You should certainly use gUPDATE OFF if you are about to write exclusively to bitmaps.

gUPDATE ON returns to normal screen updating.

gUPDATE affects anything that displays on the screen. If you are using a lot of PRINT commands, gUPDATE OFF may make a noticeable difference in speed.

Note that with gUPDATE OFF, the location of errors which occur while the procedure is running may be incorrectly reported. For this reason, gUPDATE OFF is best used in the final stages of program development, and even then you may have to remove it to locate some errors.

gUSE

Usage:
```
gUSE id%
```

Makes the drawable `id%` current. Graphics drawing commands will now go to this drawable. gUSE does not bring a drawable to the foreground (see gORDER).

gVISIBLE ON/OFF

Usage:
```
gVISIBLE ON/OFF
```

Makes the current window visible or invisible.

Raises an error if the current drawable is a bitmap.

gWIDTH

Usage:
```
width%=gWIDTH
```

Returns the width of the current drawable.

gX

Usage:
```
x%=gX
```

Returns the x current position (in from the left) in the current drawable.

gXPRINT

Usage:
```
gXPRINT string$,flags%
```

Displays `string$` at the current position, with precise highlighting or underlining. The current font and style are still used, even if the style itself is

inverse or underlined. Text mode 3 (replace) is used - both set and cleared pixels in the text are drawn.

`flags%` has the following effect:

`flags%`	*effect*
0	normal, as with gPRINT
1	inverse
2	inverse, except corner pixels
3	thin inverse
4	thin inverse, except corner pixels
5	underlined
6	thin underlined

Where lines of text are separated by a single pixel, the **thin** options maintain the separation between lines.

gXPRINT does not support the display of a list of expressions of various types.

gY

Usage:
`y%=gY`

Returns the y current position (down from the top) in the current drawable.

HEX$

Usage:
`h$=HEX$(x&)`

Returns a string containing the hexadecimal (base 16) representation of integer or long integer `x&`.

For example `HEX$(255)` returns the string `"FF"`.

Notes
To enter integer hexadecimal constants (16 bit) put a `$` in front of them. For example `$FF` is 255 in decimal. (Don't confuse this use of `$` with string variable names.)

To enter long integer hexadecimal constants (32 bit) put a `&` in front of them. For example `&FFFFF` is 1048575 in decimal.

Counting in hexadecimal is like this: `0 1 2 3 4 5 6 7 8 9 A B C D E F 10...` `A` stands for decimal 10, `B` for decimal 11, `C` for decimal 12 ... up to `F` for decimal 15. After `F` comes 10, which is equivalent to decimal 16. To understand numbers greater than hexadecimal 10, again compare hexadecimals with decimals. In these examples, 10^2 means 10x10, 10^3 means 10x10x10 and so on.

253 in decimal is:
$(2x10^2)+(5x10^1)+(3x10^0)$
$= (2x100)+(5x10)+(3x1)$
$= 200+50+3$

By analogy, &253 in hexadecimal is:
$(\&2x16^2)+(\&5x16^1)+(\&3x16^0)$
$=(2x256)+(5x16)+(3x1)$
$=512+80+3 = 595$ in decimal.

Similarly, &A6B in hexadecimal is:
$(\&Ax16^2)+(\&6x16^1)+(\&Bx16^0)$
$=(10x256)+(6x16)+(11x1)$
$=2560+96+11 = 2667$ in decimal.

You may also find this table useful for converting between hex and decimal:

hex.	*decimal*	
&1	1	$=16^0$
&10	16	$=16^1$
&100	256	$=16^2$
&1000	4096	$=16^3$

For example, &20F9 is (2x&1000)+(0x&100)+(15x&10)+9 which in decimal is: (2x4096)+(0x256)+(15x16)+9 = 8441.

All hexadecimal constants are integers ($) or long integers (&). So arithmetic operations involving hexadecimal numbers behave in the usual way. For example, `&3/&2` returns `1`, `&3/2.0` returns `1.5`, `3/$2` returns `1`.

HOUR

Usage:
```
h%=HOUR
```

Returns the number of the current hour from the system clock as an integer between 0 and 23.

IABS

Usage:
```
i&=IABS(x&)
```

Returns the absolute value, i.e. without any sign, of the integer or long integer expression `x&`.

For example `IABS(-10)` is `10`.

See also ABS, which returns the absolute value as a floating-point value.

ICON

Usage:
```
ICON name$
```

Gives the name of the bitmap file to use as the icon for an OPA.

This can only be used between APP and ENDA.

See the Advanced Topics chapter for more details of OPAs.

IF...ENDIF

Usage:
```
IF condition1
  ..
  ...
  .
ELSEIF condition2
  ...
  ....
ELSE
  ..
ENDIF
```

Does ***either***

- the statements following the IF condition

or

- the statements following one of the ELSEIF conditions (there may be as many ELSEIF statements as you like - none at all if you want)

or

- the statements following ELSE (or, if there is no ELSE, nothing at all). There may be either one ELSE statement or none.

After the ENDIF statement, the lines following ENDIF carry on as normal.

IF, ELSEIF, ELSE and ENDIF **must** be in that order.

Every IF **must** be matched with a closing ENDIF.

You can also have an IF...ENDIF structure within another, for example:

```
IF condition1
  ..
  ...
  .
ELSE
  ...
  IF condition2
  ....
  ENDIF
  ..
ENDIF
```

condition is an expression returning a logical value - for example a<b. If the expression returns logical true (non-zero) then the statements following are executed. If the expression returns logical false (zero) then those statements are ignored. For more details about logical expressions, see Appendix B.

INPUT

Usage:
INPUT *variable*
or
INPUT *log.field*

Waits for a value to be entered at the keyboard, and then assigns the value entered to a variable or data file field.

You can edit the value as you type it in. All the usual editing keys are available - the arrow keys move along the line, Esc clears the line and so on.

If inappropriate input is entered, for example a string when the input was to be assigned to an integer variable, a ? is displayed and you can try again. However, if you used TRAP INPUT, control passes on to the next line of the procedure, with the appropriate error condition being set and the value of the variable remaining unchanged.

INPUT is usually used in conjunction with a PRINT statement:

```
PROC temp:
LOCAL cent
DO
  PRINT "Centigrade?";
  INPUT cent
  PRINT "That's ";
  PRINT 32+cent*9/5,
  PRINT "fahrenheit."
  GET
UNTIL 0
ENDP
```

Note the ; at the end of the PRINT statement, used so that the cursor waiting for input appears on the same line as the message.

TRAP INPUT

If a bad value is entered (for example "abc" for a%) in response to a TRAP INPUT, the ? is not displayed, but the ERR function can be called to return the value of the error which has occurred. If the Esc key is pressed while no text is on the input line, the 'Escape key pressed' error (number -114) will be returned by ERR (provided that the INPUT has been trapped). You can use this feature to enable someone to press the Esc key to escape from inputting a value.

See also EDIT. This works like INPUT, except that it displays a string to be edited and then assigned to a variable or field. It can only be used with strings.

INT

Usage:
i&=INT(*x*)

Returns the integer (in other words the whole number) part of the

floating-point expression x. The number is returned as a long integer.

Positive numbers are rounded down, and negative numbers are rounded up. This may be undesirable - for example `INT(-5.9)` returns `-5` and `INT(2.9)` returns `2`.

In the Calculator, you need to use INT to pass a number to a procedure which requires a long integer parameter. This is because the Calculator passes all numbers as floating-point by default.

See also INTF.

INTF

Usage:
`i=INTF(x)`

Used in the same way as the INT function, but the value returned is a floating-point number. For example, `INTF(1234567890123.4)` returns `1234567890123.0`

You may also need this when an integer calculation may exceed integer range.

See also INT.

I/O functions

These functions are covered in detail in the Advanced Topics chapter.

`r%=IOOPEN(var h%,name$,mode%)`

Creates or opens a file called `name$`. Defines `h%` for use by other I/O functions. `mode%` specifies how to open the file. For unique file creation, use `IOOPEN(var h%,addr%,mode%)`

`r%=IOCLOSE(h%)`

Closes a file with the handle `h%`.

`r%=IOREAD(h%,addr%,maxLen%)`

Reads from the file with the handle `h%`. `address%` is the address of a buffer large enough to hold a maximum of `maxLen%` bytes. The value returned to `r%` is the actual number of bytes read or, if negative, is an error value.

`r%=IOWRITE(h%,addr%,length%)`

Writes `length%` bytes in a buffer at `address%` to the file with the handle `h%`.

`r%=IOSEEK(h%,mode%,var off&)`

Seeks to a position in a file that has been opened for random access. `mode%` specifies how the offset argument `off&` is to be used.

`r%=IOW(h%,func%,var a1,var a2)`

The device driver opened with handle `h%` performs the synchronous I/O function `func%` with the two further arguments.

`r%=IOA(h%,f%,var status%,var a1,var a2)`

The device driver opened with handle `h%` performs the asynchronous I/O function `f%` with two further arguments, `a1` and `a2`. The argument `status%` is set by the device driver. An IOWAIT must be performed for each IOA.

`IOWAIT`

Waits for an asynchronous I/O function to signal completion.

IOSIGNAL

Signals an I/O function's completion.

IOYIELD

Ensures that any asynchronous handler set up with IOA is given a chance to run.

IOWAITSTAT *var stat%*

Waits for an asynchronous function, called with IOA, to complete.

KEY

Usage:
```
k%=KEY
```

Returns the character code of the last key pressed, if there has been one since the last call to the keyboard. These functions count as calling to the keyboard: INPUT, EDIT, GET, GET$, KEY and KEY$.

If no key has been pressed, zero is returned.

See Appendix D for a list of special key codes. You can use KMOD to check whether *modifier keys* (Shift, Control, PSION and Caps Lock) were used.

This command does not wait for a key to be pressed, unlike GET.

KEY$

Usage:
```
k$=KEY$
```

Returns the last key pressed as a string, if there has been a keypress since the last use of the keyboard by INPUT, EDIT, GET, GET$, KEY or KEY$.

If no key has been pressed, a null string ("") is returned.

See Appendix D for a list of special key codes. You can use KMOD to check whether *modifier keys* (Shift, Control, PSION and Caps Lock) were used.

This command does not wait for a key to be pressed, unlike GET$.

KEYA

Usage:
```
err%=KEYA(var stat%,
                  var key%(1))
```

This is an asynchronous keyboard read function.

See the Advanced Topics chapter for details.

Cancel with KEYC.

KEYC

Usage:
```
err%=KEYC(var stat%)
```

Cancels the previously called KEYA function with status `stat%`.

See the Advanced Topics chapter for details.

KMOD

Usage:
```
k%=KMOD
```

Returns a code representing the state of the modifier keys (whether they were pressed or not) at the time of the

last keyboard access, such as a KEY function. The modifiers have these codes:

		binary
2	Shift down	10
4	Control down	100
8	PSION down	1000
16	Caps Lock on	10000

If there was no modifier, the function returns 0. If a combination of modifiers was pressed, the *sum* of their codes is returned - for example 20 is returned if Control (4) was held down and Caps Lock (16) was on.

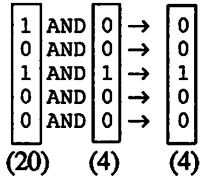
Always use immediately after a KEY/KEY$/GET/GET$ statement.

The value returned by KMOD has one binary bit set for each modifier, as shown above. By using the logical operator AND on the value returned by KMOD you can check which of the bits are set, in order to see which modifier keys were held down. For more details on AND, see Appendix B.

Example:

```
PROC modifier:
LOCAL k%,mod%
PRINT "Press a key" :k%=GET
CLS :mod%=KMOD
PRINT "Key code",k%,"with"
IF mod%=0
  PRINT "no modifier"
ENDIF
IF mod% AND 2
  PRINT "Shift down"
ENDIF
IF mod% AND 4
  PRINT "Control down"
ENDIF
IF mod% AND 8
  PRINT "Psion down"
ENDIF
IF mod% AND 16
  PRINT "Caps Lock on"
ENDIF
ENDP
```

The expression `mod% AND 4` evaluates like this if KMOD returns 20 to `mod%`:



So the expression `mod% AND 4` returns 4, that is logical 'True', so the condition `IF mod% AND 4` is met and "Control down" is shown.

LAST

Usage:
`LAST`

Positions to the last record in a data file.

LCLOSE

Usage:
`LCLOSE`

Closes the device opened with LOPEN. (The device is also closed automatically when a program ends.)

LEFT$

Usage:
`b$=LEFT$(a$,x%)`

Returns the leftmost `x%` characters from the string `a$`.

For example if `n$` has the value `Charles`, then `b$=LEFT$(n$,3)` assigns `Cha` to `b$`.

LEN

Usage:
`a%=LEN(a$)`

Returns the number of characters in `a$`.

E.g. if `a$` has the value `34 Kopechnie Drive` then `LEN(a$)` returns `18`.

You might use this function to check that a data file string field is not empty before displaying:

```
IF LEN(A.client$)
  PRINT A.client$
ENDIF
```

LN

Usage:
`a=LN(x)`

Returns the natural (base *e*) logarithm of `x`.

Use LOG to return the base 10 log of a number.

LOADM

Usage:
`LOADM module$`

Loads a translated OPL module so that procedures in that module can be called. Until a module is loaded with LOADM, calls to procedures in that module will give an error.

`module$` is a string containing the name of the module. Specify the full file name only where necessary.

Example `LOADM "MODUL2"`

Up to 4 modules can be in memory at any one time; if you try to LOADM a fifth module, you get an error. Use UNLOADM to remove a module from memory so that you can load a different one.

By default, LOADM always uses the directory of the initial running program, or the one specified by a OPA application. It is not affected by the SETPATH command.

LOC

Usage:
`a%=LOC(a$,b$)`

Returns an integer showing the position in `a$` where `b$` occurs, or zero if `b$` doesn't occur in `a$`. The search matches upper and lower case.

Example:
`LOC("STANDING","AND")` would return the value 3 because the substring AND starts at the third character of the string `STANDING`.

LOCAL

Usage:
`LOCAL variables`

Used to declare variables which can be referenced only in the current procedure. Other procedures may use the same variable names to create new variables. Use GLOBAL to declare variables common to all called procedures.

The variables may be of 4 types, depending on the symbol they end with:

- Variable names not ending with `$ % & or ()` are *floating-point variables*, for example `price, x`

- Those ending with a % are *integer variables*, for example x%, sales92%
- Those ending with an & are *long integer variables*, for example x&, sales92&.
- Those ending with a $ are *string variables*. String variable names must be followed by the maximum length of the string in brackets, for example names$(12), a$(3)

Array variables have a number immediately following them in brackets which specifies the number of elements in the array. Array variables may be of any type, for example:

```
x(6),y%(5),f$(5,12),z&(3)
```

When declaring string arrays, you must give two numbers in the brackets. The first declares the number of elements, the second declares their maximum length. For example surname$(5,8) declares five elements, each up to 8 characters long.

Variable names may be any combination of up to 8 numbers and alphabetic letters. They **must** start with a letter. The length includes the % & or $ sign, but not the () in string and array variables.

More than one GLOBAL or LOCAL statement may be used, but they **must** be on separate lines, immediately after the procedure name.

See also GLOBAL and Chapter 2.

LOCK

Usage:

```
LOCK ON
or
LOCK OFF
```

Mark an OPA (OPL application) as locked or unlocked. When an OPA is locked with LOCK ON, the System will not send it events to change files or quit. If, for example, you move onto the file list in the System screen and press Delete to try to stop that running OPA, a message will appear, indicating that the OPA cannot close down at that moment.

You should use LOCK ON if your OPA uses a command, such as EDIT or DIALOG, which accesses the keyboard. You might also use it when the OPA is about to go busy for a considerable length of time, or at any other point where a clean exit is not possible. Do not forget to use LOCK OFF as soon as possible afterwards.

'Foreground', 'background' and 'Machine on' events may still occur while the OPA is accessing the keyboard, and will be discarded.

An OPA is initially unlocked.

LOG

Usage:

```
a=LOG(x)
```

Returns the base 10 logarithm of x.

Use LN to find the base *e* (natural) log.

LOPEN

Usage:
`LOPEN device$`

Opens the device to which LPRINTs are to be sent.

No LPRINTs can be sent until a device has been opened with LOPEN.

You can open any of these devices:

- The parallel port, with `LOPEN "PAR:A"`
- The serial port, with `LOPEN "TTY:A"`
- A file on the Series 3 or an attached computer. LOPEN the file name, e.g. on a PC:

```
LOPEN "REM::C:\BAK\MEMO.TXT"
```

or on an Apple Macintosh:

```
LOPEN "REM::HD40:MY BAK:MEMO5"
```

Any existing file of the name given will be overwritten when you print to it.

Only one device may be open at any one time. Use LCLOSE to close the device. (It also closes automatically when a program finishes running.)

See Appendix C.

LOWER$

Usage:
`b$=LOWER$(a$)`

Converts any upper case characters in the string `a$` to lower case and returns the completely lower case string.

E.g. if `a$="CLARKE"`, `LOWER$(a$)` returns the string `clarke`

Use UPPER$ to convert a string to upper case.

LPRINT

Usage:
`LPRINT list of expressions`

Prints a list of items, in the same way as PRINT, except that the data is sent to the device most recently opened with LOPEN.

The expressions may be quoted strings, variables, or the evaluated results of expressions. The punctuation of the LPRINT statement (commas, semi-colons and new lines) determines the layout of the printed text, in the same way as PRINT statements.

If no device has been opened with LOPEN you will get an error.

See PRINT for displaying to the OPL window.

See LOPEN for opening a device for LPRINT.

See Appendix C for an overview of printing from OPL.

MAX

Usage:
`m=MAX(list)`
or
`m=MAX(array(),element)`

Returns the greatest of a list of numeric items.

The list can be either:

- A list of variables, values and expressions, separated by commas

or

- The elements of a floating-point array.

When operating on an array, the first argument must be the array name followed by `()`. The second argument, separated from the first by a comma, is the number of array elements you wish to operate on - for example `m=MAX(arr(),3)` would return the value of the largest of elements `arr(1)`, `arr(2)` and `arr(3)`.

mCARD

Usage:
`mCARD title$,n1$,k1%`
or
`mCARD title$,n1$,k1%,n2$,k2%`
etc.

Defines a menu. When you have defined all of the menus, use MENU to display them.

`title$` is the name of the menu. From one to six items on the menu may be defined, each specified by two parameters. The first is the item name, and the second the keycode for a hot-key. This specifies a key which, when pressed together with the PSION key, will select the option.

MEAN

Usage:
`m=MEAN(list)`
or
`m=MEAN(array(),element)`

Returns the arithmetic mean (average) of a list of numeric items.

The list can be either:

- A list of variables, values and expressions, separated by commas

or

- The elements of a floating-point array.

When operating on an array, the first argument must be the array name followed by `()`. The second argument, separated from the first by a comma, is the number of array elements you wish to operate on - for example `m=MEAN(arr(),3)` would return the average of elements `arr(1)`, `arr(2)` and `arr(3)`.

This example displays `15.0`:

```
a(1)=10
a(2)=15
a(3)=20
PRINT MEAN(a(),3)
```

MENU

Usage:
`val%=MENU`

Displays the menus defined by mINIT and mCARD, and waits for you to select an item. Returns the hot-key keycode of the item selected, as defined in mCARD, **in lower case.**

If you cancel the menu by pressing Esc, MENU returns 0.

MID$

Usage:
`m$=MID$(a$,x%,y%)`

Returns a string comprising `y%` characters of `a$`, starting at the character at position `x%`.

E.g. if `name$="McConnell"` then `MID$(name$,3,3)` would return the string `Con`.

MIN

Usage:
`m=MIN(list)`
or
`m=MIN(array(),element)`

Returns the smallest of a list of numeric items.

The list can be either:

- A list of variables, values and expressions, separated by commas

or

- The elements of a floating-point array.

When operating on an array, the first argument must be the array name followed by `()`. The second argument, separated from the first by a comma, is the number of array elements you wish to operate on - for example `m=MIN(arr(),3)` would return the minimum of elements `arr(1)`, `arr(2)` and `arr(3)`.

mINIT

Usage:
`mINIT`

Prepares for definition of menus, cancelling any existing menus. Use `mCARD` to define each menu, then `MENU` to display them.

MINUTE

Usage:
`m%=MINUTE`

Returns the current minute number from the system clock (0 to 59).

E.g. at 8.54am `MINUTE` returns `54`.

MKDIR

Usage:
`MKDIR name$`

Creates a new directory. For example, `MKDIR "M:\MINE\TEMP"` creates a `M:\MINE\TEMP` directory, also creating `M:\MINE` if it is not already there.

MONTH

Usage:
`m%=MONTH`

Returns the current month from the system clock as an integer between 1 and 12.

E.g. on 12th March 1992 `m%=MONTH` returns 3 to `m%`.

MONTH$

Usage:
`m$=MONTH$(x%)`

Converts `x%`, a number from 1 to 12, to the month name, expressed as a three-letter mixed case string.

E.g. `MONTH$(1)` returns the string `Jan`.

NEXT

Usage:
NEXT

Positions to the next record in the current data file.

If NEXT is used after the end of a file has been reached, no error is reported but the current record is a null and the EOF function returns true.

NUM$

Usage:
n$=NUM$(x,y%)

Returns a string representation of the integer part of the floating-point number x, rounded to the nearest whole number. The string will be up to y% characters wide.

- If y% is negative then the string is right-justified - for example NUM$(1.9,-3) returns " 2" where there are two spaces to the left of the 2.
- If y% is positive no spaces are added: eg NUM$(-3.7,3) returns "-4".
- If the string returned to n$ will not fit in the width y%, then the string will just be asterisks; for example NUM$(256.99,2) returns "**".

See also FIX$, GEN$, SCI$.

OFF

Usage:
OFF
or
OFF x%

Switches the Series 3 off.

When you switch back on, the statement following the OFF command is executed, for example:
OFF :PRINT "Hello again"

If you specify an integer, x%, between 8 and 16383, the machine switches off for that number of seconds and then automatically turns back on and continues with the next line of the program (16383 is about 4½ hours). However, during this time the machine may be switched on by an alarm, and of course you can turn it on with ON/Esc.

Warning: be careful how you use this command. If, due to a programming mistake, a program uses OFF in a loop, you may find it impossible to switch the Series 3 back on, and may have to reset the computer.

ONERR

Usage:
ONERR *label* or ONERR *label::*
..
..
ONERR OFF

ONERR label:: establishes an error handler in a procedure. When an error is raised, the program jumps to the label:: instead of the program stopping and an error message being displayed.

The label may be up to 8 characters long starting with a letter. It ends with a double colon (::), although you don't need to use this in the ONERR statement.

ONERR OFF disables the ONERR command, so that any errors occurring

after the ONERR OFF statement no longer jump to the label.

It is advisable to use the command ONERR OFF immediately after the `label::` which starts the error handling code.

See the Error handling chapter for full details.

OPEN

Usage:
`OPEN file$,log,f1,f2...`

Opens an existing data file `file$`, giving it the logical file name `log`, and giving the fields the names `f1`, `f2`..

You need only specify those fields which you intend to update or append, though you cannot miss out a field.

The opened file is then referred to within the program by its logical name (A, B, C or D).

Up to 4 files can be open at once.

Example:
```
OPEN "clients",A,name$,addr$
```

See also CREATE, USE and OPENR.

OPENR

This command works exactly like OPEN except that the opened file is read-only - in other words, you cannot APPEND or UPDATE the records it contains.

This means that you can run two separate programs at the same time, both sharing the same file.

OS

Usage:
`a%=OS(i%,addr1%)`
or
`a%=OS(i%, addr1%(), addr2%())`

Calls the Operating System interrupt `i%`, reading the values of all returned 8086 registers and flags. The CALL function, although simpler to use, does not allow the AL register to be passed and only the carry flag is returned, making it suitable only for certain interrupts.

The input registers are passed at the address `addr1%`. The output registers are returned at the address `addr2%` if supplied, otherwise they are returned at `addr1%`. Both addresses can be of an array, or of six consecutive integers.

Register values are stored sequentially as 6 integers and represent the register values in this order: AX, BX, CX, DX, SI and DI. The interrupt's function number, if required, is passed in AH.

The output array must be large enough to store the 6 integers returned in all cases, irrespective of the interrupt being called.

The value returned by OS is the flags register. The Carry flag, which is relevant in most cases, is in bit 0 of the returned value, so `(a% and 1)` will be 'True' if Carry is set. Similarly, the Zero flag is in bit 6, the Sign flag in bit 7 and the Overflow flag in bit 10.

For example, to find cos(pi/4):

```
PROC add:
local ax%,bx%,cx%,dx%,si%,di%
local result,cosArg,flags%
cosArg=pi/4
si%=addr(cosArg)
di%=addr(result)
ax%=$0100 REM AH=1 for cosine
flags%=os(140,addr(ax%))
```

The OS function requires *extensive* knowledge of the Operating System and related programming techniques.

See also CALL.

PARSE$

Usage:
`p$=PARSE$(f$,rel$,var off%())`

Returns a full file specification from the filename `f$`, filling in any missing information from `rel$`.

The offsets to the filename components in the returned string is returned in `off%()` which must be declared with at least 6 integers:

`off%(1)`	filing system offset (1 always)
`off%(2)`	device offset
`off%(3)`	path offset
`off%(4)`	filename offset
`off%(5)`	file extension offset
`off%(6)`	flags for wildcards in returned string

The flag values in `offset%(6)` are:
0 : no wildcards
1 : wildcard in filename
2 : wildcard in file extension
3 : wildcard in both

If `rel$` is not itself a complete file specification, the current filing system, device and/or path are used as necessary to fill in the missing parts.

`f$` and `rel$` should be separate strings.

Example:

```
p$=PARSE("NEW",
     "LOC::M:\ODB\*.ODB",x%())
```

sets `p$` to `LOC::M:\ODB\NEW.ODB` and `x%()` to `(1,6,8,13,16,0)`.

PATH

Usage:
`PATH name$`

Gives the directory to use for an OPA's files.

This can only be used between APP and ENDA.

See the Advanced Topics chapter for more details of OPAs.

PAUSE

Usage:
`PAUSE x%`

Pauses the program for a certain time, depending on the value of `x%`:

`x%`	*result*
0	waits for a key to be pressed.
+ve	pauses for `x%` twentieths of a second.
-ve	pauses for `x%` twentieths of a second or until a key is pressed.

So `PAUSE 100` would make the program pause for 100/20 = 5 seconds, and `PAUSE -100` would make the program pause for 5 seconds or until a key is pressed.

If `x%` is 0, a GET, GET$, KEY or KEY$ will return the key press which

terminated the pause. If you are not interested in this keypress, but in the one which follows it, clear the buffer after the PAUSE with a single KEY function: `PAUSE 0 :KEY`

You should be especially careful about this if `x%` is negative, since then you cannot tell whether the pause was terminated by a keypress or by the time running out.

PEEK functions

The PEEK functions find the values stored in specific bytes.

Usage:
`p%=PEEKB(x%)` returns the integer value of the byte at address `x%`

`p%=PEEKW(x%)` returns the integer at address `x%`

`p&=PEEKL(x%)` returns the long integer value at address `x%`

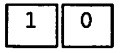
`p=PEEKF(x%)` returns the floating-point value at address `x%`

`p$=PEEK$(x%)` returns the string at address `x%`

Usually you would find out the byte address with the ADDR function. For example, if `var%` has the value 7, `PEEKW(ADDR(var%))` returns 7.

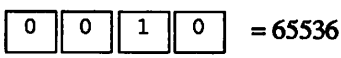
The different types are stored in different ways across bytes:

- *Integers* are stored in two bytes. The first byte is the *least significant byte*, for example:



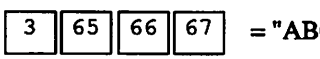
ADDR returns the address of the first (least significant) byte.

- *Long integers* are stored in four bytes, the least significant first and the most significant last, for example:



ADDR returns the address of the first (least significant) byte.

- *Strings* are stored with one character per byte, with a leading byte containing the string length, e.g.



Each letter is stored as its character code - for example, A as 65.

For example, if `var$="ABC"`, `PEEK$(ADDR(var$))` will return the string ABC. ADDR returns the address of the length byte.

- *Floating-point numbers* are stored under IEEE format, across eight bytes. PEEKF automatically reads all eight bytes and returns the number as a floating-point. For example if `var=1.3` then `PEEKF(ADDR(var))` returns `1.3`.

You can use ADDR to find the address of the first element in an array, for example ADDR(x%()). You can also specify individual elements of the array, for example ADDR(x%(2)).

See also the POKE commands and ADDR.

PI

Usage:
```
p=PI
```

Returns the value of Pi (3.14...).

POKE commands

The POKE commands store values in specific bytes.

Usage:
`POKEB x%,y%` stores the integer value `y%` (less than 256) in the single byte at address `x%`

`POKEW x%,y%` stores the integer `y%` across two consecutive bytes, with the least significant byte in the lower address, that is `x%`

`POKEL x%,y&` stores the long-integer `y&` in bytes starting at address `x%`

`POKEF x%,y` stores the floating-point value `y` in bytes starting at address `x%`

`POKE$ x%,y$` stores the string `y$` in bytes starting at address `x%`

Use ADDR to find out the address of your declared variables.

Warning: Casual use of these commands can result in the loss of data in the Series 3.

See PEEK for more details of how the different types are stored across bytes.

POS

Usage:
```
p%=POS
```

Returns the number of the current record in the current data file, from 1 (the first record) upwards.

A file can have up to 65534 records. However integers can only be in the range -32768 to +32767. Record numbers above 32767 are therefore returned like this:

record value	*returned by POS*
32767	32767
32768	-32768
32769	-32767
32770	-32766
.	.
.	.
65534	-2

To display record numbers, you can use this check:
```
IF POS<0
  PRINT 65536+POS
ELSE PRINT POS
ENDIF
```

POSITION

Usage:
`POSITION x%`

Makes record number `x%` the current record in the current data file.

If x% is greater than the number of records in the file then the EOF function will return true.

PRINT

Usage:
PRINT *list of expressions*

Displays a list of expressions on the screen. The list can be punctuated in one of these ways:

- If items to be displayed are separated by commas, there is a space between them when displayed.
- If they are separated by semi-colons, there are no spaces.
- Each PRINT statement starts a new line, unless the preceding PRINT ended with a semi-colon or comma.
- There can be as many items as you like in this list. A single PRINT on its own just moves to the next line.

Examples:
On 1st January 1993,

```
PRINT "TODAY is",
PRINT DAY;".";"MONTH;".";YEAR
```

would display `TODAY is 1.1.1993`

```
PRINT 1                 1
PRINT "Hello"           Hello
PRINT "Number",1        Number 1
```

See also LPRINT, gUPDATE.

RAD

Usage:
`r=RAD(x)`

Converts x from degrees to radians.

All the trigonometric functions assume angles are specified in radians, but it may be easier for you to enter angles in degrees and then convert with RAD.

Example:

```
PROC xcosine:
LOCAL angle
PRINT "Angle (degrees)?:";
INPUT angle
PRINT "COS of",angle,"is",
angle=RAD(angle)
PRINT COS(angle)
GET
ENDP
```

(The formula used is $^{PI*x}/_{180}$)

To convert from radians to degrees use DEG.

RAISE

Usage:
`RAISE x%`

Raises an error.

The error raised is error number x%. This may be one of the errors listed in the Error handling chapter, or a new error number defined by you.

The error is handled by the error processing mechanism currently in use - either OPL's own, which stops the program and displays an error message, or the ONERR handler if you have ONERR on.

For a full explanation, see the Error handling chapter.

RANDOMIZE

Usage:
`RANDOMIZE %x`

Gives a 'seed' (start-value) for RND.

Successive calls of the RND function produce a sequence of random numbers. If you use RANDOMIZE to set the seed back to what it was at the beginning of the sequence, the same sequence will be repeated.

For example, you might want to use the same 'random' values to test new versions of a procedure. To do this, precede the RND statement with the statement RANDOMIZE `value`. Then to repeat the sequence, use RANDOMIZE `value` again.

Example:

```
PROC SEQ:
LOCAL g$(1)
WHILE 1
  PRINT "S: set seed to 1"
  PRINT "Q: quit"
  PRINT "other key: continue"
  g$=UPPER$(GET$)
  IF g$="Q"
    BREAK
  ELSEIF g$="S"
    PRINT "Setting seed to 1"
    RANDOMIZE 1
    PRINT "First random no:"
  ELSE
    PRINT "Next random no:"
  ENDIF
  PRINT RND
ENDWH
ENDP
```

RECSIZE

Usage:
`r%=RECSIZE`

Returns the number of bytes occupied by the current record.

Use this function to check that a record may have data added to it without overstepping the 1022-character limit.

Example:

```
PROC rectest:
LOCAL name$(20)
OPEN "name",A,name$
PRINT "Enter name:",
INPUT name$
IF RECSIZE<=(1022-LEN(name$))
  A.name$=name$
  APPEND
ELSE
  PRINT "Won't fit in record"
ENDIF
ENDP
```

REM

Usage:
`REM text`

Precedes a remark you include to explain how a program works. All text after the REM up to the end of the line is ignored.

When you use REM at the end of a line you need only precede it with a space, not a space and a colon.

Examples:

```
INPUT a
b=a*.175 REM b=TAX

INPUT a
b=a*.175 :REM b=TAX
```

RENAME

Usage:
`RENAME file1$,file2$`

Renames `file1$` as `file2$`. You can rename any type of file.

You cannot use wildcards.

You can rename across directories - `RENAME"\dat\x.dbf","\n\x.dbf"` is OK. If you do this, you can choose whether or not to change the name of the file.

Example:

```
PRINT "Old name:" :INPUT a$
PRINT "New name:" :INPUT b$
RENAME a$,b$
```

REPT$

Usage:
`r$=REPT$(a$,x%)`

Returns a string comprising `x%` repetitions of `a$`.

For example, if `a$="ex"`, `r$=REPT$(a$,5)` returns `exexexexex`.

RETURN

Usage:
`RETURN` or `RETURN variable`

Terminates the execution of a procedure and returns control to the point where that procedure was called (ENDP does this automatically).

`RETURN variable` does this as well, but also passes the value of `variable` back to the calling procedure. The variable may be of any type. You can return the value of any single array element - for example `RETURN x%(3)`. *You can only return one variable.*

`RETURN` on its own, and the default return through ENDP, causes the procedure to return the value 0 or a null string.

Example:

```
PROC price:
  LOCAL x
  PRINT "Enter price:",
  INPUT x
  x=tax:(x)
  PRINT x
  GET
ENDP

PROC tax:(price)
  RETURN price+17.5%
ENDP
```

RIGHT$

Usage:
`r$=RIGHT$(a$,x%)`

Returns the rightmost `x%` characters of `a$`.

Example:

```
PRINT "Enter name/reference",
INPUT c$
ref$=RIGHT$(c$,4)
name$=LEFT$(c$,LEN(c$)-4)
```

RMDIR

Usage:
`RMDIR str$`

Removes the directory given by `str$`. You can only remove empty directories.

RND

Usage:
`r=RND`

Returns a random floating-point number in the range 0 (inclusive) to 1 (exclusive).

To produce random numbers between 1 and `n` - e.g. between 1 and 6 for a dice - use the following statement:
`f%=1+INT(RND*n)`

RND produces a different number every time it is called within a program. A fixed sequence can be generated by using RANDOMIZE. You might begin by using RANDOMIZE with an argument generated from MINUTE and SECOND (or similar), to seed the sequence differently each time.

Example:

```
PROC rndvals:
LOCAL i%
PRINT "Random test values:"
DO
  PRINT RND
  i%=i%+1
  GET
UNTIL i%=10
ENDP
```

SCI$

Usage:

```
s$=SCI$(x,y%,z%)
```

Returns a string representation of x in scientific format, to y% decimal places and up to z% characters wide.

For example:

```
SCI$(123456,2,8)="1.23E+05"
SCI$(1,2,8)="1.00E+00"
SCI$(1234567,2,-9)=" 1.23E+06"
```

If the number does not fit in the width specified then the returned string contains asterisks.

See also FIX$, GEN$, SCI$.

SCREEN

Usage:

```
SCREEN width%,height%
```

or

```
SCREEN width%,height%,x%,y%
```

Changes the size of the window in which text is displayed. x%, y% specify the character position of the top left corner; if they are not given, the text window is centred in the screen.

An OPL program can initially display text to the whole screen.

SECOND

Usage:

```
s%=SECOND
```

Returns the current time in seconds from the system clock (0 to 59).

E.g. at 6:00:33 SECOND returns 33.

SECSTODATE

Usage:

```
SECSTODATE s&,var yr%,
var mo%,var dy%,var hr%,
var mn%,var sc%,var yrday%
```

Sets the variables passed by reference to the date corresponding to s&, the number of seconds since 00:00 on 1 January 1970. yrday% is set to the day in the year (1-366).

s& is an **unsigned** long integer. To use values greater than +2,147,483,647, subtract 4,294,967,296 from the value.

See also DATETOSECS, HOUR, MINUTE, SECOND, dDATE, DAYS.

SETNAME

Usage:
SETNAME *name$*

Sets the name of the running OPA to name$ and redraws any status window, using that name below the icon.

SETPATH

Usage:
SETPATH *name$*

Sets the current directory for file access - for example, SETPATH "a:\docs". LOADM continues to use the directory of the initial program, or the one specified by a OPA application, but all other file access will be to the new directory.

SIN

Usage:
s=SIN(*angle*)

Returns the sine of angle, an angle expressed in radians.

To convert from degrees to radians, use the RAD function.

SPACE

Usage:
s&=SPACE

Returns the number of free bytes on the device on which the current (open) data file is held.

Example:

```
PROC stock:
OPEN "A:stock",A,a$,b%
WHILE 1
  PRINT "Item name:";
  INPUT A.a$
  PRINT "Number:";
  INPUT A.b%
  IF RECSIZE>SPACE
    PRINT "Disk full"
    CLOSE
    BREAK
  ELSE
    APPEND
  ENDIF
ENDWH
ENDP
```

SQR

Usage:
s=SQR(x)

Returns the square root of x.

STATUSWIN

Usage:
STATUSWIN ON
or
STATUSWIN OFF

Makes a permanent status window visible or invisible.

The permanent status window is behind all other OPL windows. In order to see it, you must use SCREEN and gSETWIN to reduce the size of the text and graphics windows. You should ensure that your program does not create windows over the top of it.

STD

Usage:
`s=STD(list)`
or
`s=STD(array(),element)`

Returns the standard deviation of a list of numeric items.

The list can be either:

- A list of variables, values and expressions, separated by commas

or

- The elements of a floating-point array.

When operating on an array, the first argument must be the array name followed by `()`. The second argument, separated from the first by a comma, is the number of array elements you wish to operate on - for example `m=STD(arr(),3)` would return the standard deviation of elements `arr(1)`, `arr(2)` and `arr(3)`.

This function gives the sample standard deviation, using the formula: $SQR(\Sigma(x_i-\bar{x})^2/(n-1))$ where $\bar{x}$ means $\Sigma x_i/n$. To convert to population standard deviation, multiply the result by $SQR((n-1)/n)$.

STOP

Usage:
`STOP`

Ends the running program.

SUM

Usage:
`s=SUM(list)`
or
`s=SUM(array(),element)`

Returns the sum of a list of numeric items.

The list can be either:

- A list of variables, values and expressions, separated by commas

or

- The elements of a floating-point array.

When operating on an array, the first argument must be the array name followed by `()`. The second argument, separated from the first by a comma, is the number of array elements you wish to operate on - for example `m=SUM(arr(),3)` would return the sum of elements `arr(1)`, `arr(2)` and `arr(3)`.

TAN

Usage:
`t=TAN(angle)`

Returns the tangent of `angle`, an angle expressed in radians.

To convert from radians to degrees, use the DEG function.

TESTEVENT

Usage:
`t%=TESTEVENT`

Returns 'True' if an event has occurred, otherwise returns 'False'.

The event is not read by TESTEVENT - it may be read with GETEVENT.

TRAP

Usage:
`TRAP command`

TRAP is an error handling command. It may precede any of these commands:

Data file commands
APPEND, UPDATE
BACK, NEXT, LAST, FIRST, POSITION
USE, CREATE, OPEN, OPENR, CLOSE, DELETE

File commands
COPY, COMPRESS, ERASE, RENAME
LOPEN, LCLOSE
LOADM, UNLOADM

Directory commands
MKDIR, RMDIR

Data entry commands
EDIT, INPUT

Graphics commands
gSAVEBIT, gCLOSE, gUSE
gUNLOADFONT, gFONT, gPATT, gCOPY

For example, `TRAP FIRST`.

Any error resulting from the execution of the command will be trapped. Program execution will continue at the statement after the TRAP statement, but ERR will be set to the error code.

TRAP overrides any ONERR.

See the Error handling chapter for further details.

TYPE

Usage:
`TYPE num%`

Sets the type of an OPA, from 0 to 4, with `num%`.

This can only be used between APP and ENDA.

See the Advanced Topics chapter for more details of OPAs.

UNLOADM

Usage:
`UNLOADM module$`

Removes from memory the module `module$` loaded with LOADM.

`module$` is a string containing the name of the translated module.

The procedures in an unloaded module cannot then be called by another procedure.

UNTIL

See DO.

UPDATE

Usage:
`UPDATE`

Deletes the current record in the current data file and saves the current field values as a new record at the end of the file.

This record, now the last in the file, remains the current record.

Example:

```
A.count=129
A.name$="Brown"
UPDATE
```

Use APPEND to save the current field values as a new record.

UPPER$

Usage:

```
u$=UPPER$(a$)
```

Converts any lower case characters in a$ to upper case, and returns the completely upper case string.

Example:

```
...
CLS :PRINT "Y to continue"
PRINT "or N to stop."
g$=UPPER$(GET$)
IF g$="Y"
  nextproc:
ELSEIF g$="N"
  RETURN
ENDIF
...
```

Use LOWER$ to convert to lower case.

USE

Usage:

```
USE logical name
```

Selects the data file with the *logical name* A, B, C or D. The file must previously have been opened with OPEN, OPENR or CREATE and not yet be closed.

All the record handling commands (such as POSITION or UPDATE) then operate on this file.

USR

Usage:

```
u%=USR(pc%,ax%,bx%,cx%,dx%)
```

Executes your machine code, returning an integer. The USR code (i.e. the assembler code you have written) **must return with a far RET**, otherwise the program will crash.

The values of of ax%, bx%... are passed to the AX, BX... 8086 registers. The microprocessor then executes the machine code starting at pc%. At the end of the routine, the value in the AX register is passed back to u%.

WARNING: Casual use of this function can result in the loss of data in the Series 3.

This example shows a simple operation, ending with a far RET:

```
PROC trivial:
LOCAL t%(2),u%,ax%
  t%(1)=$c032   REM xor al,al
  t%(2)=$cb     REM retf
  ax%=$1ab
  u%=usr(addr(t%(1)),ax%,0,0,0)
  REM returns (ax% AND $FF00)
  PRINT u% REM 256 ($100)
  GET
ENDP
```

See also USR$, ADDR, PEEK, POKE.

USR$

Usage:

```
u$=USR$(pc%,ax%,bx%,cx%,dx%)
```

Executes your machine code, returning a string. The USR$ code you have written **must return with a far RET**, otherwise the program will crash.

The values of of ax%, bx%... are passed to the ax, bx... 8086 registers.

The microprocessor then executes the machine code starting at `pc%`. At the end of the routine, the value in the ax register must point to a length-byte preceded string. This string is then copied to `u$`.

WARNING: Casual use of this function can result in the loss of data in the Series 3.

See USR for an example. See also ADDR, PEEK, POKE.

VAL

Usage:
`v=VAL(numeric string)`

Returns the floating-point number corresponding to a numeric string.

The string must be a valid number - e.g. not `"5.6.7"` or `"196f"`. Expressions, such as `"45.6*3.1"`, are not allowed. Scientific notation such as `"1.3E10"`, is OK.

E.g. `VAL("470.0")` returns `470.0`

See also EVAL.

VAR

Usage:
`v=VAR(list)`
or
`v=VAR(array(),element)`

Returns the variance of a list of numeric items.

The list can be either:

- A list of variables, values and expressions, separated by commas

or

- The elements of a floating-point array.

When operating on an array, the first argument must be the array name followed by `()`. The second argument, separated from the first by a comma, is the number of array elements you wish to operate on - for example `m=VAR(arr(),3)` would return the variance of elements `arr(1)`, `arr(2)` and `arr(3)`.

This function gives the sample variance, using the formula: $\Sigma(x_i-\bar{x})^2/(n-1)$ where $\bar{x}$ means $\Sigma x_i/n$. To convert to population variance, multiply the result by `(n-1)/n`

VECTOR

Usage:
```
VECTOR I%
label1,label2,...
labelN
ENDV
```

VECTOR `I%` jumps to label number `i%` in the list - if `i%` is 1 this will be the first label, and so on. The list is terminated by the ENDV statement. The list may spread over several lines, with a comma separating labels in any one line but no comma at the end of each line.

If `i%` is not in the range 1 to N, where N is the number of labels, the program continues with the statement after the ENDV statement.

See also GOTO.

WEEK

Usage:
```
w%=WEEK(day%,month%,year%)
```

Returns the week number in which the specified day falls, as an integer between 1 and 53.

`day%` must be between 1 and 31, `month%` between 1 and 12, `year%` between 1900 and 2155.

Each week is taken to begin on the 'Start of week' day, as specified in Time. When a year begins on a different day, it counts as week 1 if there are four or more days before the next week starts..

WHILE...ENDWH

Usage:
```
WHILE expression
  ..
  ...
  ..
  ....
ENDWH
```

Repeatedly performs the set of instructions between the WHILE and the ENDWH statement, so long as `expression` returns logical true - non-zero.

If `expression` is not true, the program jumps to the line after the ENDWH statement.

Every WHILE must be closed with a matching ENDWH.

See also DO...UNTIL and Chapter 3.

YEAR

Usage:
```
y%=YEAR
```

Returns the current year as an integer between 1900 and 2155 from the system clock.

For example, on 5th May 1992 `YEAR` returns `1992`.

Appendices

A

Summary for users with OPL experience

OPL has evolved from the Psion Organiser II through versions for the MC and HC computers to the version built into the Series 3. This appendix explains the changes made with each new implementation.

For more details of the following topics, look them up in the index or table of contents.

Bear in mind that some OPL keywords return or allow different values according to screen size and keyboard layout.

Note: the comparisons against HC and MC OPL in this appendix assume the original versions of these computers. Some of the Series 3 OPL features mentioned may be incorporated into future releases in the HC and MC ranges.

Using OPL on the SERIES 3

To create an OPL module, press the System button, and move the cursor to the Program icon. (Alternatively, hold down PSION and press the Program button.) Select 'New file' from the 'File' menu, give the name to use, and then type in the module. The editing features are the same as in Word, but the 'Word' menu has been replaced by a 'Prog' menu, which has new programming options. Select its 'Translate' option to translate the module, then the 'Run' option to run it. You can also run the translated module from the System screen, as its name automatically appears under a RunOpl icon - move the cursor right to the RunOpl icon, then down onto the module name, and press Enter.

Existing OPL modules are listed under the Program icon on the System screen. To edit one, move the cursor onto the module name and press Enter.

To stop a running program, press PSION-Esc.

The Series 3 screen is 240 points wide by 80 points deep. The PRINT command can display 40 columns by 9 rows of text.

OPL programs can be translated as applications which appear in the System screen. This is done with the APP keyword.

Error numbers/messages are similar to those of the HC/MC, with some additions.

Files

Translated modules are compatible between the Series 3, HC and MC computers. You can build code libraries which can be used on all three machines, bearing in mind differences between screen size, location of files etc. Note, though, a module translated on a Series 3 will only run on a HC or MC if it does not use keywords unique to Series 3 OPL.

Procedures translated on the Organiser need re-translation, as they will not run on the Series 3.

The Series 3 uses the same MS-DOS filing system as the MC, although for everyday use it 'hides' the directories from you. The directories and file extensions used are:

Type of file	Directory	File extension
OPL modules	\OPL	.OPL
translated modules	\OPO	.OPO
data files	\OPD	.ODB
bitmaps	\OPD	.PIC

Changes from Organiser to MC OPL

Procedures and modules

More than one procedure may be stored in a single .OPL file, or *module*. The beginning of each procedure is identified by the line PROC *procedure name*: and the end of each procedure by the line ENDP. So a module might look like this:

```
PROC oneroot:
   LOCAL x
   PRINT "Type a number:"
   INPUT x
   root:(x)
   GET
ENDP

PROC root:(p)
   CLS
   PRINT "Root of",p,"is",sqr(p)
ENDP
```

When you run the module, the first procedure in the module is executed. Any other procedures in the module are available to be called by the first procedure.

Procedures can be called from other modules only if you've loaded the modules with the new LOADM command. Up to 4 modules can be held in memory at any one time. Use UNLOADM to remove a module from memory.

Other changes

Data files now have a special format. Records can be a maximum of 1022 bytes long.

The FIND function compares a search string agaist entire text fields. To find text anywhere within text fields, add an asterisk at either end of the string - e.g., `FIND("*JONES*")`.

When you use REM at the end of a line you need only precede it with a space, not a space and a colon. So these two lines are equivalent:

```
RAISE -37 :REM disk full

RAISE -37 REM disk full
```

No calculator memories are available to OPL on the MC, although they are on the Series 3.

The specification of the BEEP command has changed.

Different effects are produced by values below 32, used with `PRINT CHR$()`. See the Character Set appendix for more details.

New functions and commands:

- SCREEN allows you to set the maximum size of the text window.
- LOPEN must now be used before LPRINTing, to specify the destination of the LPRINTs. Use LCLOSE if you then want to LPRINT to another device. You can also use LOPEN to print to a file.
- COMPRESS copies data files, and makes sure that erased records aren't copied to the destination device.
- FINDW, DIRW$, COPYW and DELETEW on Organiser model LZ have been renamed FIND, DIR$, COPY and DELETE - the functions/commands which previously had these names having been removed.

These functions/commands have been removed from OPL: CLOCK, UDG, DISP, VIEW, FREE, MENU and MENUN. Also, the Comms Link commands are not available.

Long integers

A new variable type has been added - *long integers*, specified by adding an `&` symbol to the variable name instead of the `%` symbol, e.g. `price&`, `x&(5)`.

Long integers are 32- rather than 16-bit signed integers, giving the range +2,147,483,647 to -2,147,483,648 rather than +32767 to -32768.

Long hexadecimal constants should have an `&` instead of a `$` sign in front of them. Even when the constant fits into an integer, you can use `&` to widen the constant to 32 bits.

Advanced use

POKEL, POKEF and POKE$ have been added to POKEB and POKEW, and PEEKL, PEEKF and PEEK$ have been added to PEEKB and PEEKW. This means you can poke and peek the full range of value types - strings, long integers etc.

There is a range of *I/O file handling facilities*, allowing you to open, read, write and position within any type of file.

To complement the keypress functions, KMOD allows you to detect any modifiers, such as Control, which have been pressed.

Changes from MC to HC OPL

Graphics commands

Many graphics commands are available. They can, for example:

- Draw lines and boxes.
- Fill areas with patterns.
- Display text in a variety of styles, at any position on the screen.
- Scroll areas of the screen.
- Manipulate up to eight separate windows and bit patterns.
- Read data back from the screen.

The TRAP command can be used with several new commands.

The CURSOR command can be still be used to switch the text cursor on and off, but can also define the shape and position of a graphics cursor.

The gUPDATE command affects anything that displays on the screen. It may make a noticeable difference in speed if, for example, you are using a lot of PRINT commands.

Other changes

When a program runs, it is given the full screen on which to display text. Use the SCREEN command to define a different window size for text display.

The SCREEN command can position the text window anywhere on the screen. You can also use SCREEN repeatedly to change the size of the window.

A new function, OS, allows you to call any interrupt service in the Operating System, and read all returned registers and flags.

GETEVENT and TESTEVENT give details of system events of any kind.

Changes from HC to SERIES 3 OPL

Everyday features

Many of the interface features of the Series 3 applications - for example menus, dialog boxes, shadowed boxes and status windows - are available via OPL keywords.

Several keywords have been added for OPAs (OPL applications).

The two colons at the end of the label in ONERR or GOTO are optional. The colons must still be used where the label itself occurs in the program.

You can pause a running program by pressing PSION-S. It will be paused as soon as it next tries to display something on the screen. Press PSION-S again to resume running the program.

The EVAL function evaluates a string and returns the result as a floating-point number.

The VECTOR command allows you to jump to one of a list of labels. This can save cumbersome IF statements.

Procedures may now be called using a string expression for the procedure name.

BEEP may be used with the duration made negative. It will first check whether the sound system is in use (perhaps by another OPL program), and return if it is. This stops BEEP waiting until the sound system is free.

There are new graphics keywords - gINVERT, gXPRINT, gBORDER and gCLOCK.

DATETOSECS and SECSTODATE convert between standard system dates / times and the number of seconds since midnight on 1/1/1970.

The calculator memories M0 to M9 are available to OPL.

Advanced features

MKDIR makes a new directory, RMDIR removes a directory, and SETPATH sets the current directory for file access.

The command line arguments of the OPL program you run, including its full pathname, are returned by CMD$.

Two new I/O commands, IOWAITSTAT and IOYIELD, have been added.

PARSE$ will produce a single full file specification from two separate parts, or break one down into its constituent parts.

B

Operators and logical expressions

Operators

These operators are available in OPL:

Arithmetic operators

+	add
-	subtract
*	multiply
/	divide
**	raise to a power
-	unary minus (in negative numbers - for example, `-10`)
%	percent

Comparison operators

>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
=	equal to
<>	not equal to

Logical and bitwise operators

AND
OR
NOT

The % operator

The percentage operator can be used in expressions like this:

`60+5%` - i.e. 60 plus 5% of 60. Result: `63`
`60-5%` - i.e. 60 minus 5% of 60. Result: `57`
`60*5%` - i.e. 5% of 60. Result: `3`
`60/5%` - i.e. what number is 60 5% of. Result: `1200`.

It can also be used like this:

`105>5%` - i.e. what number, when increased by 5%, becomes 105. Result: `100`
`105<5%` - i.e. how much of 105 was a 5% increase. Result: `5`

Examples

To add 15% tax to 345:

```
345+15%      Result = 396.75
```

To find out what the price was before tax:

```
396.75>15% Result = 345
```

To find out how much of a total price is tax:

```
396.75<15% Result = 51.75
```

Precedence

Highest: **

- (unary minus) NOT

* /

+ -

= > < <> >= <=

Lowest: AND OR

So 7+3*4 returns 19 (3 is first multiplied by 4, and 7 is added to the result) not 40 (4 times 10).

When there's equal precedence

In an expression where all operators have equal precedence, they are evaluated from left to right (with the exception of powers). For example, in a+b-c, a is added to b and then c is subtracted from the result.

Powers are evaluated from right to left - for example, in a%**b%**c%, b% will first be raised to the power of c% and then a% will be raised to the power of the result.

Changing precedence with brackets

The result of an expression such as a+b+c is the same whether you first add a to b, or b to c. But how is a+b*c/d evaluated? You may want to use brackets to either:

- Make it obvious what the order of calculation is

or

- Change the order of calculation.

By default, a+b*c/d is evaluated in the order: b multiplied by c, then divided by d, then added to a. To perform the addition and the division before the multiplication, use brackets: (a+b)*(c/d). When in doubt, simply use brackets.

Precedence of integer and floating-point values

You are free to mix floating-point and integer values in expressions, but be aware how OPL handles the mix:

- In expressions, integers linked by an operator to a floating-point number are converted to floating-point

but finally

- The evaluted result of the right hand side of an expression is automatically converted to the same type as the variable to which it is assigned.

For example, your procedure might include the expression `a%=b%+c` This is handled like this: `b%` is converted to floating-point and added to `c`. The resulting floating-point value is then automatically converted to an integer in order to be assigned to the integer variable `a%`.

Such conversions may produce odd results - for example `a%=3.0*(7/2)` makes `a%=9`, but `a%=3.0*(7.0/2)` makes `a%=10`. OPL does not report this as an error, so it's up to you to ensure that it doesn't happen - unless you want it to.

Type conversions and rounding down

There are three numeric types - floating-point, integer and long integer. You can assign any of these types to any other. The value on the right hand side will be automatically converted to the type of the variable on the left hand side. For example:

- If you assign an integer value to a floating-point variable, there are no problems.
- If you assign a floating-point value to an integer variable, the value is converted to an integer, always rounded **towards zero** - for example, if you declare `LOCAL c%` and then say `c%=3.75`, the value `3.75` is converted to the value `3`.

Rounding-down can sometimes cause unusual results. For example, `a%=2.9` would give `a%` the value 2, and `a%=-2.3` would give `a%` the value –2.

When you run a module, if the left hand side of an assignment has a narrower range than the right hand side, you may get an error (for example, if you had `x%=a&` where `a&` had the value 320000).

To control how floating-point numbers are rounded when converted, use the INT function.

Logical expressions

The comparison operators and logical operators are based on the idea that a certain situation can be evaluated as either *true* or *false*. For example, if `a%=6` and `b%=8`, `a%>b%` would be 'False'.

These operators are useful for setting up alternative paths in your procedures. For example you could say:

```
IF salary<expenses
   PRINT "bad"
   doA:
ELSE
   PRINT "good"
   doB:
ENDIF
```

You can also make use of the fact that the result of these logical expressions is represented by an integer:

- 'True' is represented by the integer -1
- 'False' is represented by the integer 0 (zero).

	E.g.	*Result returned*	*Return value*
<	a<b	True if a less than b	-1
		False if a greater than or equal to b	0
>	a>b	True if a greater than b	-1
		False if a less than or equal to b	0
<=	a<=b	True if a less than or equal to b	-1
		False if a greater than b	0
>=	a>=b	True if a greater than or equal to b	-1
		False if a less than b	0
<>	a<>b	True if a not equal to b	-1
		False if a equal to b	0
=	a=b	True if a equal to b	-1
		False if a not equal to b	0

These integers can be assigned to a variable or displayed on the screen to tell you whether a particular condition is true or false, or used in an IF statement.

For example, in a procedure you might arrive at two sub-totals, a and b. You want to find out which is the greater. So use the statement, `PRINT a>b`. If zero is displayed, a and b are equal or b is the larger number; if -1 is displayed, a>b is true - a is the larger.

Logical and bitwise operators

The operators AND, OR and NOT have different effects depending on whether they are used with floating-point numbers or integers:

When used with floating-point numbers...
... AND, OR and NOT are *logical operators*, and have the following effects:

Example	*Result*	*Integer returned*
a AND b	True if both a and b are non-zero	-1
	False if either a or b are zero	0
a OR b	True if either a or b is non-zero	-1
	False if both a and b are zero	0
NOT a	True if a is zero	-1
	False if a is non-zero	0

When used with integer or long integer values...
... AND, OR and NOT are *bitwise operators.*

The way OPL holds integer numbers internally is as a binary code - 16-bit for integers, 32-bit for long integers. Bitwise means that an operation is performed on individual bits. A bit is *set* if it has the value 1, and *clear* if it has the value 0. Long integer values with AND, OR and NOT behave the same as integer values.

AND *Sets the result bit if both input bits are set, otherwise clears the result bit.*

For example, the statement `PRINT 12 AND 10` displays 8. To understand this, write 12 and 10 in binary:

```
12      0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0
10      0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0
```

AND acts on each pair of bits. Thus, working from left to right - discounting the first 12 bits (since 0 AND 0 gives 0):

```
1 AND 1  →      1
1 AND 0  →      0
0 AND 1  →      0
0 AND 0  →      0
```

The result is therefore the binary number 1000, or 8.

OR *Sets the result bit if either input bit is set, otherwise clears the result bit.*

What result would the statement `PRINT 12 OR 10` give? Again, write down the numbers in binary and apply the operator to each pair of digits:

```
1 OR 1  →  1
1 OR 0  →  1
0 OR 1  →  1
0 OR 0  →  0
```

The result is the binary number 1110, or 14 in decimal.

NOT *Sets the result bit if the input bit is not set, otherwise clears the result bit.*

`NOT` works on only one number. It returns the *one's complement*, i.e. it replaces 0s with 1s and 1s with 0s.

So since 7 is 0000000000000111, `NOT 7` is 1111111111111000. This is the binary representation of -8.

A quick way of calculating `NOT` for integers is to add 1 to the original number and reverse its sign. So `NOT 23` is -24, `NOT 0` is -1 and `NOT -1` is 0.

C

Printing from OPL

You can use LPRINT in an OPL program to print to any of these devices:

- A parallel port - "PAR:A"
- A serial port - "TTY:A"
- A file on the Series 3 or an attached computer.

The advanced page formatting and font control features of the Series 3 cannot be used by OPL.

To a parallel printer

Physically connect the printer.

In your OPL program, set up the port with the statement LOPEN "PAR:A".

Provided the port is not already in use, the connection is now ready. LPRINT will print on the attached printer.

Example

```
PROC prints:
   OPEN "clients",A,a$
   LOPEN "PAR:A"
   PRINT "Printing..."
   DO
      IF LEN(A.a$)
         LPRINT A.a$
      ENDIF
      NEXT
   UNTIL EOF
   LPRINT CHR$(12); :LCLOSE
   PRINT "Finished" :GET
ENDP
```

To a serial printer

Physically connect the printer.

In your OPL program, set up the port with the statement `LOPEN "TTY:A"`.

Now LPRINT should print to the printer. If it does not, the serial port parameters are not correct.

Serial port parameters

`LOPEN "TTY:A"` opens the serial port with the following default characteristics:

9600 baud
no parity
8 data bits
1 stop bit
RTS handshaking.

- If your printer *does* match these characteristics, the LOPEN statement establishes the connection with the printer, and subsequent LPRINT statements will print there successfully.
- If your printer *does not* match these characteristics, you must use a procedure like the one listed below to change the characteristics of the serial port, before LPRINTs will print successfully to your printer.

Printers very often use DSR (DSR/DTR) handshaking, and you may need to set the port to use this.

Setting the serial port characteristics

Calling the procedure
The `rsset:` procedure listed below provides a convenient way to set up the serial port.

Each time you use an LOPEN "TTY: " statement, follow it with a call to the `rsset:` procedure. Otherwise the LOPEN will use the default characteristics.

Passing values to the procedure
Pass the procedure the values for the five port characteristics, like this:

```
rsset:(baud%,parity%,data%,stop%,hand%)
```

To find the value you need for each characteristic, use the tables below. You *must* give values to all five parameters, in the correct order.

Baud =	50	75	110	134	150	300	600	1200
value =	1	2	3	4	5	6	7	8
	1800	2000	2400	3600	4800	7200	9600	
	9	10	11	12	13	14	15	

Parity =	NONE	EVEN	ODD
value =	0	1	2

Data bits = 5, 6, 7 or 8

Stop bits = 2 or 1

Handshaking =	ALL	NONE	XON	RTS	XON+RTS
value =	11	4	7	0	3
	DSR	XON+DSR		RTS+DSR	
	12	15		8	

For example, call rsset: like this:

```
rsset:(8,0,8,1,0)
```

1200 baud, no parity, 8 data bits, 1 stop bit, RTS/CTS

The rsset: procedure

```
PROC rsset:(baud%,parity%,data%,stop%,hand%)
   LOCAL frame%,srchar%(6),dummy%,err%
   frame%=data%-5
   IF stop%=2 :frame%=frame% OR 16 :ENDIF
   IF parity% :frame%=frame% OR 32 :ENDIF
   srchar%(1)=baud% OR (baud%*256)
   srchar%(2)=frame% OR (parity%*256)
   srchar%(3)=(hand% AND 255) OR $1100
   srchar%(4)=$13
   err%=IOW(-1,7,srchar%(1),dummy%)
   IF err% :RAISE err% :ENDIF
ENDP
```

Take care to type this program in exactly as it appears here.

Example of calling the procedure

```
PROC test:
   PRINT "Testing port settings"
   LOPEN "TTY:A"
   LOADM "rsset"
   rsset:(8,0,8,1,0)
   LPRINT "Port OK" :LPRINT
   PRINT "Finished" :GET
   LCLOSE
ENDP
```

Advanced use

The section of the `rsset:` procedure which actually sets the port is this:

```
srchar%(1)=baud% OR (baud%*256)
srchar%(2)=frame% OR (parity%*256)
srchar%(3)=(hand% AND 255) OR $1100
srchar%(4)=$13
err%=IOW(-1,7,srchar%(1),dummy%)
IF err% :RAISE err% :ENDIF
```

The four elements of the array `srchar%` contain the values specifying the port characteristics. If you want to write a shorter procedure, you could work out what these values need to be for a particular setup you want, assign these values to the elements of the array, and then use the IOW function (followed by the error check) exactly as above.

To a file

Printing to a file on a PC or Apple Macintosh

As if you were going to transfer a file:

- Physically connect the Series 3 and the other computer.
- Select the 'Remote link' option in the System screen and press Enter.
- Run the server program (MCLINK or MACLINK, for example) on the other computer.

In your OPL program, specify the destination file with an LOPEN statement. For example, to a PC:

```
LOPEN "REM::C:\BACKUP\PRINTOUT\MEMO.TXT"
```

Any subsequent LPRINT would go to the file MEMO.TXT in the directory \BACKUP\PRINTOUT on the PC's drive C:.

With a Macintosh, you might use a file specification like this:

```
LOPEN "REM::HD40:MY BACKUP:PRINTED:MEMO5"
```

An LPRINT would now go to the file MEMO5 in the PRINTED folder, itself in the MY BACKUP folder on the hard drive HD40. Note that colons are used, to separate the various parts of the file specification.

Printing to a file on the SERIES 3

In your OPL program, specify the destination file with an LOPEN statement like this:

```
LOPEN "B:\PRINT\MEMO.TXT"
```

This would send each subsequent LPRINT to the file MEMO.TXT in the \PRINT\ directory on an SSD in drive B:.

D

Character codes

To find out a character's character code either look up the character in the table given in the back of your User Guide, or press the Calc button and type the % sign followed by the character - for example %P returns 80. Characters with codes from 0 to 127 are the same as in the ASCII character set. Codes 128 to 255 are compatible with the IBM code page 850.

Codes from 256 upwards are for keys unique to the Series 3 - see the list below.

Character codes of special keys

The GET and KEY functions return the character code of the key that was pressed. Some of the keys are not in the character set. They return these numbers:

Esc	27	TAB	9
DELETE	8	Enter	13

Special keys

↑	256	↓	257
→	258	←	259
PAGE UP	260	PAGE DOWN	261
HOME	262	END	263
MENU	290	HELP	291

The PSION key adds 512 to the value of the key pressed. For example, PSION-a is 609 (512+97), and PSION-Help (Dial) is 803 (512+291).

Special character codes with PRINT

These values can be used with PRINT and CHR$():

7 beep
8 backspace
9 tab
10 line feed
12 form feed (clear screen)
13 carriage return (cursor to left of window)

For example, `PRINT CHR$(8)` moves the cursor backwards, one character to the left.

Index

!

symbol 130
% operator 32, 236
& symbol 231
? prompt 24
@ symbol 47

A

ABS function 168
ACOS function 168
ADDR function 168
ALERT function 90, 168
AND operator 241
APP keyword 145, 168, 229
APPEND command 169
Apple Macintosh
    file specifications 128
Applications
    installing 229
    See also OPAs
Arguments 32
    conversion 155
Arithmetic operators 20, 236
Array variable 16
ASC function 169
ASIN function 169
Assign, value to variable 19
AT command 169
ATAN function 170
Auto-indentation 8

B

BACK command 58, 170
Baseline of text 67
Baud rate 245
Beep
    with PRINT 250
BEEP command 170
Bitmaps 150
    filenames 128, 229
    files 75
    in memory 75
Bitwise operators 241
Bold text 67
    while editing 8
Borders
    in windows 71
Branches 31
BREAK command 34, 171
BUSY command 92, 171
'Busy' messages 92

C

Calculator memories 26
CALL function 171, 211
Calling procedures 41
Character codes
    special keys 250
    with GET,GET$,KEY,KEY$ 25
CHR$ function 171
Clearing pixels 65-66, 68
Clearing rectangles
    text window 139
Clock 76
CLOSE command 54, 172
CLS command 172
CMD$ function 172
Co-ordinates 63
Commands
    See individual command names
Comparison operators 236
COMPRESS command 172, 231
Conditions, in loops 33
Constants 21

CONTINUE command 173
Control-Program button 10, 12
Control-Shift-Program keypress 12
Conversion, of types 239
COPY command 173, 231
'Copy' option
in System screen 11
Copying modules 11
COS function 174
COUNT function 174
CREATE command 52, 174
Current position 63
Current window 70
Cursor
in graphics 67, 73
user-defined 76
CURSOR command 67, 73, 174, 232
Cursor movement
with PRINT 250
Cursor position
reading 139
See also gAT
See also gMOVE

D

Data bits 245
Data file
appending/updating 56
checking for EOF 57
closing 54
copying 172, 231
creating in OPL 52
example program 103
field name 52
filenames 60, 128, 229
finding a record 59
logical name 52
moving between records 58
opening 53
ordering 105
structure 51
using a different file 57
Database
and OPL data files 60
Date input 86
DATETOSECS function 175
DATIM$ function 175
DAY function 175
DAYNAME$ function 175
DAYS function 175
dBUTTONS command 90, 176
dCHOICE command 86, 176
dDATE command 86, 177
'Declaration error' 17
Declaring variables
examples 18
explained 15
LOCAL and GLOBAL 46
dEDIT command 85, 177
Default window 70
DEG function 177
DELETE command 178, 231
'Delete file' option
in System screen 11
Deleting modules 11
dFILE command 85, 178
dFLOAT command 86, 179
DIALOG function 84, 179
Dialogs 84
choice lists 86
date/time input 86
exit keys 90
number input 86
string display 89
string input 85
dINIT command 84, 179
DIR$ function 179, 231
Directories 127-128, 229

Directory, in filenames 127
Division
problems 22
dLONG command 86, 180
DO...UNTIL command 29, 180
Dots
drawing 64
Double-height text 68
DOW function 180
dPOSITION command 90, 181
Drawables 75
Drives
in filenames 127
dTEXT command 89, 181
dTIME command 86, 181
dXINPUT command 85, 182

E

EDIT command 182
EDIT with TRAP 116
End of file, in data file 57
ENDA keyword 145
ENDP 5
ENDV keyword 35, 223
EOF function 57, 183
ERASE command 58, 183
ERR function 116, 183
ERR$ function 116, 183
Error handling
ERR, ERR$ 116
ONERR 117
overview 113
RAISE 119
TRAP 115
Error messages 229
listed 121
with ERR/ERR$ 116
Error numbers, listed 121
Errors
common syntax errors 112
while running 113
Esc key, in INPUT 116
ESCAPE ON/OFF command 184
EVAL function 184, 233
Events 146, 189, 220
EXIST function 184
EXP function 184
Expression 20
See also EVAL function
EXT keyword 145, 184
Extensions, in filenames 127

F

False, and true 240
Fields
in data files 51
input to 54
types 52
Filenames 128, 229
extensions 127
See also dFILE
Filing system 127
FIND function 59, 185, 230-231
FIRST command 58, 185
FIX$ function 185
Floating-point variables
explained 16
precedence 239
range 16
rounded down 239
FLT function 185
Fonts 67
Text window 139
user-defined 76
See also gFONT
Functions
See individual function names

G

gAT command 64, 186
gBORDER command 71, 186
gBOX command 65, 186
gCLOCK command 76, 186
gCLOSE command 72, 187
gCLS command 187
gCOPY command 75, 187
gCREATE function 70, 187
gCREATEBIT function 75, 188
GEN$ function 188
GET function 188
    special key codes 250
GET$ function 188
GETCMD$ function 146, 189
GETEVENT command 146, 189
gFILL command 65, 190
gFONT command 67, 190
gGMODE command 66, 190
gHEIGHT function 190
gIDENTITY function 190
gINFO command 190
gINVERT command 191
GIPRINT command 92, 191
gLINEBY command 63, 191
gLINETO command 64, 192
gLOADBIT function 192
gLOADFONT function 76, 192
GLOBAL command 46, 192
Global variables
    returning values 47
    'Undefined externals' error 46, 48
gMOVE command 63, 193
gORDER command 72, 193
gORIGINX function 194
gORIGINY function 194
GOTO command 35, 194, 233
gPATT command 66, 75, 194
gPEEKLINE command 194
gPOLY command 76, 194
gPRINT command 67, 195
gPRINTB command 195
gPRINTCLIP function 196
gRANK function 196
Graphics
    text window 76
gSAVEBIT command 150, 196
gSCROLL command 196
gSETWIN command 196
gSTYLE command 68, 197
gTMODE command 68, 197
gTWIDTH function 197
gUNLOADFONT command 76, 197
gUPDATE command 75, 198
gUSE command 70, 198
gVISIBLE command 72, 198
gWIDTH function 198
gX function 198
gXPRINT command 198
gY function 199

H

Handshaking 245
HC
    differences in OPL 228
HEX$ function 199
Hexadecimal 199
Hot-keys 81-82
HOUR function 87, 200

I

I/O functions
    device handling 136
    error handling 129
    example program 135
    opening a file 131
    overview 129

positioning in a file 134
reading a file 133
writing to a file 134
IABS function 200
ICON keyword 145, 150, 200
Icons 150
IDs
for fonts 76
for windows 70
IF...ENDIF command 31, 200
'Indentation' option 8
Information messages 92
INPUT
to data fields 54
INPUT command 23, 201
INPUT with TRAP 116
INT function 201
'Integer overflow' 21
Integer variables
precedence 239
range 16
INTF function 202
Inverse text 68
Inverting pixels 65-66, 68
IOA function 136
IOOPEN function 131
IOREAD function 133
IOSEEK function 134
IOSIGNAL function 137
IOW function 136, 139
IOWAIT function 137
IOWAITSTAT function 137
IOWRITE function 134
IOYIELD function 137

K

KEY function 203
special key codes 250
KEY$ function 203
KEYA function 138
KEYC function 138
Keypresses, recognising 25
KMOD function 203

L

Labels 35, 233
LAST command 58, 204
LCLOSE command 204
LEFT$ function 204
LEN function 205
LN function 205
LOADM command 41, 205
LOC function 205
LOC:: in full filenames 127
LOCAL command 15, 46, 205
LOCK command 149, 206
LOG function 206
Logical expressions 240
Logical name
of data file 52
Logical operators 241
Long integer
range 16
variable 16
Loops
conditions 33
IF...ENDIF 31
maximum nested 31
LOPEN
a filename 248
PAR:A parallel port 244
TTY:A serial port 245
LOPEN command 207
LOWER$ function 207
LPRINT command 207, 231

M

M0 - M9 26
MAX function 207
MC
    differences in OPL 228
mCARD command 81, 208
MEAN function 208
'Memory full' 9-10
MENU function 81-82, 208
Menu key 12, 91
Menus 81
    problems 82
MID$ function 208
MIN function 209
mINIT command 81, 209
MINUTE function 87, 209
MKDIR command 128, 209
Modules
    calling other modules 41
    containing several procedures 40
    copying 11
    defined 5
    deleting 11
    editing 8
    names 6
    running 10
    saving 94
    stopping while running 11
    translating 9
    unloading 42
    See also procedures
MONTH function 209
MONTH$ function 209

N

Names
    of variables 17
'New file' option
    in Program editor 11
    in System screen 6, 11
NEXT command 58, 210
NOT operator 241
NUM$ function 210
Number input 23, 86
Numbers
    See Floating-point, Integer, Long integer

O

OFF command 210
ONERR command 117, 210, 233
OPAs 144
OPEN command 53, 211
'Open file' option
    in Program editor 11
OPENR command 211
Operating System 171, 211
Operators
    arithmetic 20
    bitwise 241
    listed 236
    logical 240-241
    precedence 238
    with integer values 241
OR operator 241
Ordering a data file 105
Organiser
    differences in OPL 228
OS function 211
Out of memory 82
'Outline' option 8
Overwriting
    in graphics 65-66, 68

P

PAR:A device 244
Parallel printing
    PAR: device 244
Parameters
    explained 43
    multiple 44
    returning values 44
    'type mismatch' error 43
    types 43
Parity 245
PARSE$ function 128, 212
Passwords
    on OPL files 8
    See dXINPUT
PATH keyword 145, 212
PAUSE command 212
Pausing a program 11, 233
PEEK functions 213, 231
Percentage operator 236
PI function 214
Pixels 63
Plain text 135
POKE commands 214, 231
Ports
    See Serial port, Parallel port
POS function 58, 214
POSITION command 58, 214
Precedence (of operators) 238
PRINT command 22, 215, 229
    special characters 250
Printing
    data file 105
    rsset: procedure 245
    serial port settings 245
    to parallel port PAR: 244
    to serial port TTY: 245
PROC - explained 7
Procedures
    calling 41, 47, 233
    creating 6
    defined 5
    in other modules 41, 230
    saving 94
    translating 9
    See also modules
'Prog' menu 8
'Program' icon 11
Programs 5
    See also modules, procedures
Proportional font
    while editing 8
PSION key 83

R

RAD function 215
RAISE command 119, 215
Random numbers 100
RANDOMIZE command 216
Range
    floating-point 16
    integer/long integer 16
Records
    explained 51
    finding 59
    moving between 58
    saving to file 56
RECSIZE function 216
REM command 216, 230
REM:: in full filenames 127
RENAME command 216
REPT$ function 217
RETURN command 44, 217
RIGHT$ function 217
RMDIR command 128, 217
RND function 217
rsset: procedure 245
'Run' option

in Program editor 10
Running a module 10

S

'Save as template' option 8
Saving modules 94
SCI$ function 218
SCREEN command 218, 231-232
Screen positions 63
Screen size 229
SECOND function 218
SECSTODATE command 218
Serial port
changing settings 245
default settings 245
Serial printing
TTY: device 245
SETNAME command 91, 149-150, 219
SETPATH command 128, 219
'Show error' option 8
SIN function 219
Sorting a data file 105
SPACE function 219
SQR function 219
Statement, defined 8
Status window 12, 91
STATUSWIN command 91, 150, 219
STD function 220
Stop bits 245
STOP command 36, 220
Stop, running module 11
Stopwatch 106
String
input 23, 85
joining (concatenating) 22
variable 16
'Structure fault' 112
Style
text window 139
See also gSTYLE
SUM function 220
'Syntax error' 9, 112
System screen
OPAs 144

T

Tab width 8
Tabstops 250
TAN function 220
Template files
in OPL 8
TESTEVENT function 146, 220
Text input 23, 85
Time
current time 87
stopwatch 106
Time input 86
'Too complex' 31
'Too many items' 91
'Too wide' 91
'Translate' option
in Program editor 9
Translating
explained 9
TRAP command 58, 115, 221, 232
TRAP with INPUT/EDIT 116
True, and false 240
TTY:A device 245
Type conversion 239
TYPE keyword 145, 150, 221
'Type mismatch' error 43

U

'Undefined externals' error 46, 48
Underlined text 68
UNLOADM command 42, 221
UPDATE command 221

UPPER$ function 222
USE command 57, 222
USR function 222
USR$ function 222

V

VAL function 223
VAR function 223
Variables
    array 16
    assign value to 19
    declaring 15
    declaring - examples 18
    explained 15
    floating-point 16
    GLOBAL v LOCAL 46
    long integer 16
    names 17
    operations upon 20
    string 16
    types - explained 15
VECTOR command 35, 223, 233

W

WEEK function 224
WHILE...ENDWH command 30, 224
Wildcards
    in data file search 59
Windows 70
    borders 71
    current window 70
    default window 70
    hiding 72
    information about 73
    overlapping 72
    text window 76

Y

YEAR function 224

Version 2.0
Nov 91
Part no. 6103-0002-02
English